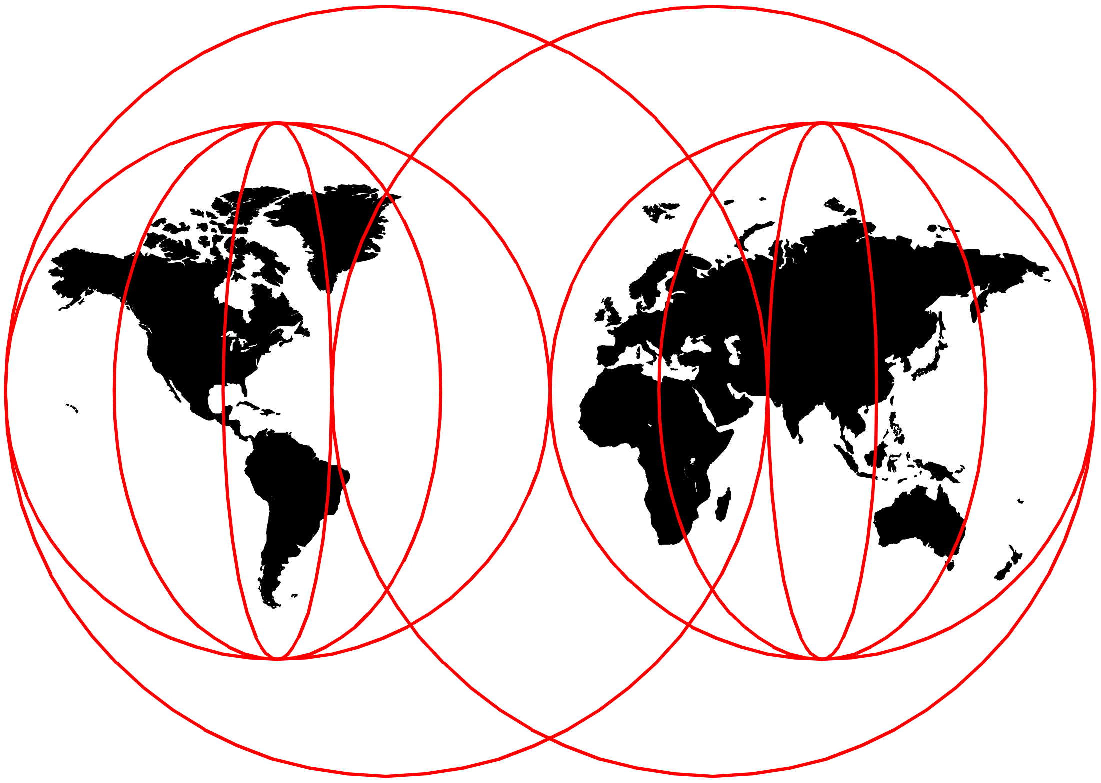
Getting to Know VisualAge C++ Version 4.0

Stephen Kurlow, Camille Pagee



International Technical Support Organization

www.redbooks.ibm.com

SG24-5489-00

International Technical Support Organization

SG24-5489-00

Getting to Know VisualAge C++ Version 4.0

February 2000

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix B, “Special notices” on page 99.

First Edition (February 2000)

This edition applies to Version 4.0 of VisualAge C++, Program Number 30L8360, for use with the IBM OS/2, Windows NT, or IBM AIX Operating Systems.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. QXXE Building 80-E2
650 Harry Road
San Jose, California 95120-6099

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Preface

This redbook will accelerate your learning of VisualAge C++ Version 4.0 for use with IBM OS/2, Windows NT, or IBM AIX Operating systems, by taking you on a journey to work through a typical small software project. The structure of this small software project is detailed in Chapter 2, “About the sample project” on page 7.

This small software project consists of several components. A couple of executables will be progressively built as the book unfolds. The complexity will increase to introduce the execution of external tools that will generate C++ files that will need to be compiled and linked into the executables. A dynamic link library (DLL) contains the C++ source code that will use these executables to build the DLL. A static library contains the C++ source code. A comparison to building this as a DLL is included. A resource library contains Windows resources such as window definitions and externalized strings. A static library, upon which the first DLL will depend, will be used as an example of a technique to enable greater development efficiency.

First, we provide an introduction to concepts and basic syntax of a configuration file, with examples. This will be especially useful for those who are new to VisualAge C++ Version 4.0 and are starting to learn about configuration files. Then we discuss how the structure of your projects, and the way you work with them, may change, once you have made the initial transition to VisualAge C++.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization San Jose Center.

Stephen Kurlow is a Software Engineer at the Business Integration Services group of IBM Global Services Australia. He designs and builds Object-Oriented and Client/Server applications for the IBM OS/2, AIX and Microsoft Windows NT platforms using IBM VisualAge C++. He has acquired a Bachelor of Engineering degree in Computer Systems Engineering at the University of Technology, Sydney.

Camille Pagee is an Information Developer at the IBM Toronto Lab. She was part of the documentation team for VisualAge C++ Version 4 and for IBM C and C++ Compilers for OS/2 and Windows, Version 3.6. At the time of publication, she is working on online help for the next release. She holds a degree in Translation and a certificate in Technical and Professional Writing from York University.

Thanks to the following people for their invaluable contributions to this project:

Joe DeCarlo, Project Leader
International Technical Support Organization, San Jose Center

Derek Inglis, Compiler Development
IBM Toronto Lab

Robert Klarer, Compiler Development
IBM Toronto Lab, Toronto, Canada

Martin Lansche, VisualAge TecTeam
IBM Toronto Lab, Toronto, Canada

Dwayne Moore, Customer Support Representative
IBM Toronto Lab, Toronto, Canada

Paul Pacholski, Worldwide Sales Support for VisualAge C++
IBM Toronto Lab, Toronto, Canada

Silvio Zarb, Senior Analyst Programmer
Software Solutions ANZ, Asset Finance, Melbourne, Australia

Comments welcome

Your comments are important to us!

We want our redbooks to be as helpful as possible. Please send us your comments about this or other redbooks in one of the following ways:

- Fax the evaluation form found in “IBM Redbooks evaluation” on page 111 to the fax number shown on the form.
- Use the online evaluation form found at `http://www.redbooks.ibm.com/`
- Send your comments in an Internet note to `redbook@us.ibm.com`

Part 1. Getting acquainted with VisualAge C++ Version 4.0

In the following eleven chapters, we introduce the basics of configuration files, and contrast them with makefiles. You will learn the syntax of the most common configuration file directives, and see examples of doing common build tasks.

If you are already very comfortable with configuration files, you may find the "Other useful references" sections at the end of each chapter helpful.

Chapter 1. The leap to VisualAge C++ Version 4.0

If you have heard little about VisualAge C++ Version 4.0, then you have likely heard at least that it incorporates a new incremental compiler. This represents a significant change in the way software developers build their source code. For the most part, Version 4.0 represents change in the following areas:

- Incremental compilation and linking: this means compiling only those granular parts, such as C++ methods, that have changed or been affected by change since the previous build. This is in contrast to traditional batch compilers that compile larger parts, such as entire files, that have changed or been affected by the timestamp change of dependent files.
- Configuration files define what source code to build instead of makefiles which define what steps to take and how to build your source code.
- A database called the *codestore* contains all project information—compiled code, source code, and options—and eliminates the need to maintain file dependencies, or generate object files.
- Earlier error reporting of C++ coding errors is provided.
- Richer OpenClass library reduces the number of classes you need to develop.
- You can write C++ code using the latest C++ standard.
- Templates are automatically instantiated (there is no need for a tempinc directory).
- Loosely coupled tools such as the compiler, linker, LPEX editor and the debugger are replaced with an Integrated Development Environment (IDE).

No matter how you look at it, change requires effort. In this book, our intention is to reduce the effort of adapting to the new environment. If you have already made the switch, and are finding that you are not maximizing the promised benefits of ease of maintenance and improved build times, this book may help you refine your techniques and get the most out of your investment.

1.1 Incremental compilation and linking

Build time is the major complaint of C++ programmers. With the advent of VisualAge C++, the time required for both the initial build and subsequent rebuilds of an application is reduced by as much as 10 times, depending on the size of the application. Larger improvements are typically obtained with larger applications.

With the right configuration, incremental builds are significantly faster than a traditional batch compiler because the compiler and linker only work on what has changed and leaves the rest as-is. If an existing function's content is changed, then only that function is recompiled and relinked. The cost of change is proportional to its impact on your base of source code.

1.2 What? No makefiles?

Configuration file is not just a new name for a makefile. A makefile is a set of instructions that tell a compiler what to do, and in what order. The incremental compiler in VisualAge C++ does not see your build process as a set of steps.

Instead, it views your source files and outputs in terms of relationships. A configuration file is more like a family tree than a set of procedures. The differences can have a big impact on how efficiently the project is built. That is why it is important not only to develop configuration files that will replace your makefiles, but to go further and develop *good* configurations for your project.

For the most part, it is simpler to produce a configuration file, and easier to make it work, than to do the same with a makefile. The major differences between a configuration file and a makefile are summarized in Table 1.

Table 1. How makefiles differ from configuration files

Subject	Makefile	Configuration file
Source files	What they are.	Same.
Source files	Which compiler to use.	Not necessary for you to specify.
Object files	How to compile each source file into an object file.	Object files do not need to be built.
Linking	How to link object files into a DLL or executable.	Object files do not need to be linked.

1.3 The codestore

The first time you build a project, a codestore is created. Every function, declaration or macro that is compiled is tracked in this database and identified in terms of its impact on other functions. When a function changes, VisualAge C++ already knows which other functions, if any, will be affected. There are several effects of using this system:

- It no longer matters which files contain which functions; since only the changed functions and the functions they affect will be recompiled, you do not need to tell the compiler which files are dependent on each other.
- Intermediate object files are not required. If you have told the compiler which sources to use, and what output you need, it will figure out how to generate the target (executable, DLL, and so on) without the intermediate files. However, if for any reason your build process needs an object file, then VisualAge C++ can also be instructed to generate them.
- All the information required for debugging is already contained in the codestore, so you never have to set options for generating traditional debug information with C++ source files.
- Similarly, the codestore eliminates the need to include header files, provide forward declarations, and prevent inclusion of the same header file more than once. VisualAge C++ can determine the dependencies between your declarations and definitions, and it knows the order in which to compile them.
- You do not have to reinvoke the compiler or set any options in order to see preprocessor output.
- Templates are automatically instantiated, and form part of the codestore. A temporary directory such as *tempinc* is not needed.

1.4 Earlier error reporting

Each time a traditional batch compiler completes the compilation of a target (in a makefile context), the information about each source file is discarded. When you rebuild, the entire process is repeated in the same order. Since header files are normally processed first, you can wait a long time before your own code is interpreted.

Due to incremental compilation and linking, only the source code you change and the functions it impacts (if any) are compared to the information saved from the last build. You do not have to wait for included header files and previously compiled code to be reprocessed before build errors are found.

1.5 The latest C++ standard

All of the great features in the latest ISO C++ standard are now implemented in VisualAge C++. This enables you to write portable applications that use features like namespaces and dynamic casting. Earlier C++ compilers on IBM OS/2, IBM AIX and Windows NT platforms supported earlier C++ standards and they did not all support the same standard. This made it difficult to write portable applications.

1.6 Reusable libraries—Standard Template Library and OpenClass

VisualAge C++ now provides the Standard Template Library (STL) as well as OpenClass.

OpenClass library has grown enormously from previous versions of VisualAge C++ and now numbers approximately 1000 classes. This allows you to reuse many more classes to improve your productivity as a C++ programmer. Also OpenClass library is now highly portable between IBM OS/2, IBM AIX, and Windows NT platforms, making it easier to develop portable applications.

When choosing a library to use, you should be aware that the two libraries are incompatible and only one of the two can be used. Also bear in mind your target platforms. If you are using non-IBM C++ compilers, then do not use OpenClass, as it is only available for OS/2, AIX, Windows NT, OS/400 and OS/390.

For more information on the Standard Template Library and incompatibilities with IBM Open Class, read the Migration User's Guide and Reference for VisualAge C++ Professional, Version 4.0, which is available at:

```
www.software.ibm.com/ad/visualage_c++/downloads.htm
```

1.7 Integrated Development Environment

The IDE provides you with one place to work, within which you can perform the following and much more:

- Find classes and inspect their members.
- Produce a class hierarchy of all classes or a subset of them.
- Edit your source code.
- Edit your configuration files.

- Build your source code.
- Design your user interfaces.
- Search for text in all files incorporated during the build.
- Find uses of classes and their members.

Chapter 2. About the sample project

The concepts discussed in this book will be illustrated by following the development path for a sample project that reflects a real-life application development scenario. No source code will be used, as the intent of the book is to focus on the configuration files and their relationship to makefiles.

Figure 1 shows an overview of the way the project application was built.

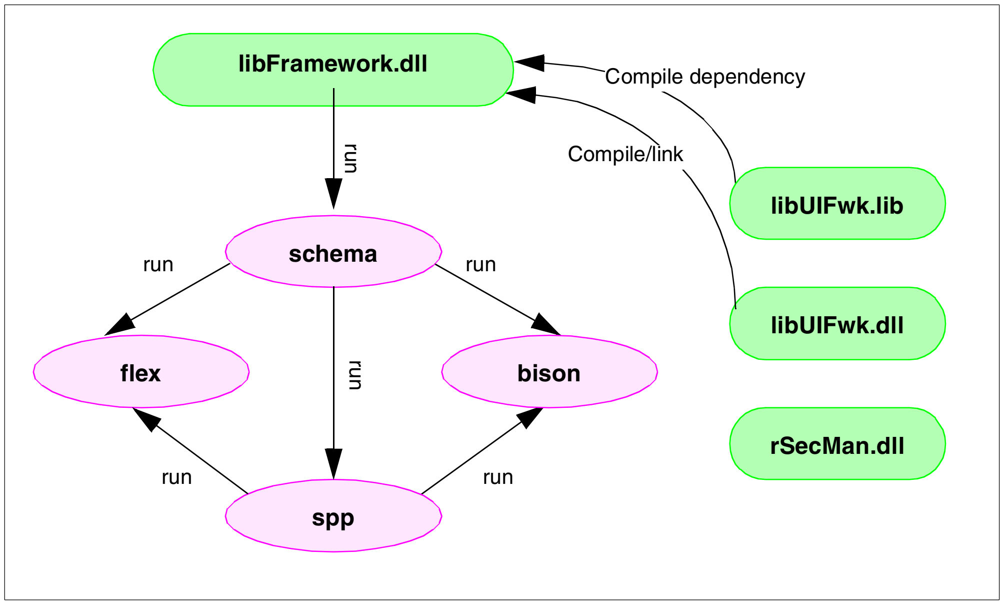


Figure 1. Outline of our project

The structure of this small software project can be described as follows:

- Executables *spp.exe* and *schema.exe* that use two other external tools named *bison.exe* and *flex.exe* to build themselves. The external tools take non-C++ files as input and generate C++ source files to be compiled as part of spp.exe and schema.exe.
- A dynamic link library (DLL) named *libFramework.dll* that **initially** has no link dependencies on other components in our project. The executable *schema.exe* is used in a similar fashion to the above external tools to take non-C++ files and generate C++ source files to be compiled as part of *libFramework.dll*. Later in this book you will find this DLL split into two libraries to illustrate a technique to enable greater development efficiency.
- A static library named *libUIFwk.lib* that has a compile dependency on *libFramework.dll* because *libUIFwk.lib* uses header files from *libFramework.dll* in order to build *libUIFwk.lib*. The static library *libUIFwk.lib* can also be built as a DLL, when it will then also have a link dependency on *libFramework.dll*. The pros and cons of developing the library either way will be discussed.
- A resource library named *rSecMan.dll* that has no link dependencies on other components. It contains Windows resources such as window definitions and externalized strings.

This book will follow the development of configuration files for several of these components, and illustrate how the configurations develop at each step to accommodate dependencies and the need for tools other than the compiler:

- In Chapter 3 we will create a basic configuration to build spp.exe.
- In Chapter 4 we will show how options are added to the build for spp.exe.
- In Chapter 5 we will begin optimizing the configuration.
- In Chapter 6 we will introduce user-defined variables.
- In Chapter 7 we will insert a step to run an external tool to generate additional source files, which will also be added to the configuration before compilation. At the end of this chapter, we will have a completed configuration for building spp.exe.
- In Chapter 8, once the tools are completed, we will build *libFramework.dll*, which runs *schema* during the build. The generated target will be directed to a different working directory.
- In Chapter 9, we will build the static library libUIFwk.lib.
- In Chapter 10 we will illustrate building the resource DLL *rSecMan.dll*.
- In Chapter 11, we will have two targets in a configuration, in order to illustrate conflicts caused by the *One-Definition Rule*, and how to avoid them.

Chapter 3. An introduction to configuration files

The first step in building our sample project is to create the tool spp.exe, which is needed to produce the source files that will be needed as input for later stages in development.

3.1 Targets

A configuration file is constructed around the intended outputs, or *targets*, of your build. A target can be an executable, a dynamic link library, static library, shared library, or object file. (In most cases, you will find that you no longer need to generate object files, but we will go into more detail on this later.) In the configuration file, the definition of your target looks like this:

```
target "targetname"
{
}
```

That is all! To have a target in a configuration file, you must have:

1. The *target directive*, which consists of the word **target** plus the name of your target, in quotation marks.
2. Open and close braces ({ and }), in which you will list your sources.

So, the first step in developing the configuration for building spp.exe is to define the target, like this:

```
target "spp.exe"
{
}
```

3.2 Sources

The open and close braces following the target directive will contain your source files. Source files can be placed outside the scope of a target directive. However, if a source file does not appear between the { and } of a target directive, it will not be linked as part of that target. Source files that appear outside the scope of a target directive will only be compiled. The best practice is to place your source files within the scope of the target they belong to.

Here is how the configuration file looks when you add a source file to the project:

```
target "targetname"
{
   source "source_file_name"
}
```

In other words, you have added the source file to the project by adding a *source directive* to the target directive. The source directive consists of the word **source** plus your source file name, in quotation marks (one directive can also contain a list of source file names, separated by commas).

There are nine source files needed to build spp.exe. Three of them, SppParser.cpp, SppParser.hpp, and SppScanner.cpp, are to be generated by external tools (that is, they do not exist yet). However, we know that they will be required as source files. If we insert all of the files into the target directive, the configuration file now looks like this:

```
target "spp.exe"
{
   source "Tools\schema\CommandParser.cpp"
   source "Tools\schema\CommandParser.hpp"
   source "Tools\schema\Preprocessor.cpp"
   source "Tools\schema\Preprocessor.hpp"
   source "iaset.inl"
   source "iisetavl.c"
   //These three files do not exist yet. We will create them later:
   source "SppParser.cpp"
   source "SppParser.hpp"
   source "SppScanner.cpp"
}
```

The filenames alone are sufficient if the files are located in one of the directories listed in the INCLUDE environment variable.

However, when this configuration is built, we will get the following error messages:

```
CPPC1919E:The file "SppParser.cpp" is not found.
CPPC1919E:The file "SppParser.hpp" is not found.
CPPC1919E:The file "SppScanner.cpp" is not found.
CPPC1930E:An extension to handle source "iaset.inl" cannot be found.
```

There will also be other error messages, which we will address in later sections.

The first three messages appear because the files have not been created yet. We will later add a step to generate these files. For the present, we can prevent this message by commenting out the source files with C++-style comments.

The fourth message (CPPC1930) means that the compiler does not recognize the file extension .inl, and does not know how to proceed. To tell the compiler that the file is C++ code, we have to add a *type clause* to the directive.

Similarly, although the file iisetavl.c is a C++ file containing template definitions, the compiler will identify the .c extension as belonging to a C file. If we want the code to be processed as C++, we have to add the type clause. (Even for C files, C++ recompilation is much faster after the initial build because the C compiler is not incremental).

The configuration for building spp.exe now looks like this:

```
target "spp.exe"
{
   source "Tools\schema\CommandParser.cpp",
      "Tools\schema\CommandParser.hpp",
      "Tools\schema\Preprocessor.cpp",
      "Tools\schema\Preprocessor.hpp"
   //This is a C++ file, to be inlined:
   source type ("cpp") "iaset.inl"
   //This is a template C++ file:
   source type ("cpp") "iisetavl.c"
   //These three files do not exist yet. We will create them later:
   //source "SppParser.cpp"
   //source "SppParser.hpp"
```

```
      //source "SppScanner.cpp"
   }
```

Multiple source files can be specified with one source directive. That is, we could also have listed all of the source files of the same type after the word **source** by separating the file names with commas.

3.3 If you are migrating

The configuration described above involves both the compiling and linking phases of building: the target directive indicates that linking is necessary, and this phase is invoked automatically.

In certain situations, you may not want to invoke the link phase. You can prevent the link phase from being invoked by commenting out the target. For example:

```
   //target "spp.exe"
   {
      source "Tools\schema\CommandParser.cpp"
      source "Tools\schema\CommandParser.hpp"
      //
      //other sources, etcetera...
      //
   }
```

In this example, the sources will still be compiled, so you can see if there were any compilation errors and address them before going on to the link phase. This is advisable when migrating code from projects developed on an older version of VisualAge C++.

3.4 A different route to compilation

Following is the equivalent part of the makefile that was used for this stage of our project (we have shown here only the targets and dependencies in the makefile, for the sake of clearer correspondence to the concepts in this chapter: the full text is printed later, in Chapter 7, “Running external tools” on page 31.)

```
# This stanza loosely corresponds to the target directve:
.\NT\bin\spp.exe:
.\NT\obj\Tools\Schema\Preprocessor.obj \
.\NT\obj\Tools\Schema\SppParser.obj \
.\NT\obj\Tools\Schema\SppScanner.obj \
.\NT\obj\Tools\Schema\CommandProcessor.obj
@echo Linking  .\NT\bin\spp.exe
.
.
.\NT\obj\Tools\Schema\Preprocessor.obj:  \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools\Schema\Preprocessor.cpp \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools/Schema/Preprocessor.hpp \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools/Schema/CommandProcessor.hpp
.
.
.\NT\obj\Tools\Schema\SppParser.obj:  \
.\NT\obj\Tools\Schema\SppParser.hpp
.\NT\obj\Tools\Schema\sppParser.cpp
.
```

```
.
.\NT\obj\Tools\Schema\SppScanner.obj:  \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\SppParser.hpp
.\NT\obj\Tools\Schema\SppScanner.cpp
.
.\NT\obj\Tools\Schema\CommandProcessor.obj:  \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools\Schema\CommandProcessor.cpp\
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools/Schema/CommandProcessor.hpp
.
.
# cleaning up the build environment:
del /Q /F .\NT\obj\Tools\Schema\Preprocessor.obj
del /Q /F .\NT\obj\Tools\Schema\SppParser.obj
del /Q /F .\NT\obj\Tools\Schema\SppScanner.obj
del /Q /F .\NT\obj\Tools\Schema\CommandProcessor.obj
```

As you can see, several major differences between the steps used by VisualAge C++ and those used by the traditional compiler are already apparent:

- Where the configuration file described only the final target (the .exe file), the makefile describes every stage along the way. Specific instructions for generating first an object file, then an executable, are needed. See Figure 2 and Figure 3.

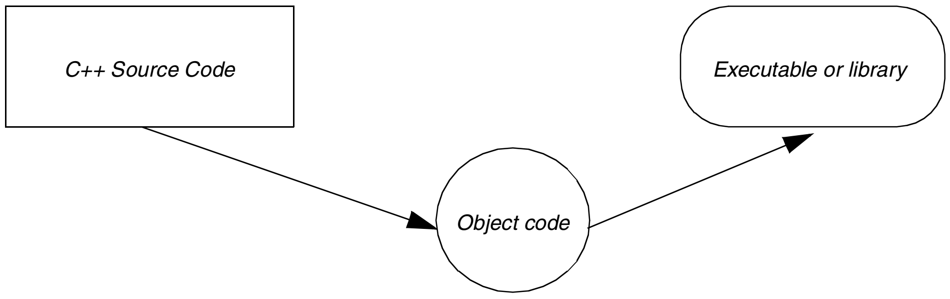


Figure 2. Traditional route to producing an executable or library

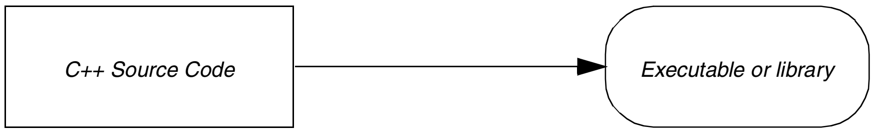


Figure 3. How VisualAge C++ Version 4.0 produces an executable or library

- Since the traditional compiler creates object files which are no longer needed after compilation, a cleanup phase has to be executed to delete all the unwanted files.

3.5 Other useful references

Refer to the online help for the full syntax of the source and target directives, as well as for the following:

- "Types of Source Files", a list of file extensions recognized by the compiler.
- "C Compilation: Comparing C and C++ Builds", which contains more information on the difference between processing C source and C++ source.
- "Data Elements in Configuration Files", which has more detailed information on how strings (for example, file paths) are interpreted in a configuration file.

Chapter 4. Adding options

In the previous section, we developed a basic configuration consisting of the sources and target required to build an executable, spp.exe:

```
target "spp.exe"
{
   source "Tools\schema\CommandParser.cpp"
   source "Tools\schema\CommandParser.hpp"
   source "Tools\schema\Preprocessor.cpp"
   source "Tools\schema\Preprocessor.hpp"
   //This is a C++ file, to be inlined:
   source type ("cpp") "iaset.inl"
   //This is a template C++ file:
   source type ("cpp") "iisetavl.c"
   //These three files do not exist yet. We will create them later:
   //source "SppParser.cpp"
   //source "SppParser.hpp"
   //source "SppScanner.cpp"
}
```

In this section, we will add options for linking and for macro processing.

4.1 How options work

An option directive takes a form similar to a source or target directive:

```
   option option_type(option_name,setting)
   {
   }
```

For example, the option to determine which version of the runtime is linked is:

```
   link(linkWithSharedLib, yes)
```

The option directive applies only to the sources that appear between the { and }. You can include several options in a single directive, separated by commas. The directive can be nested within other directives. The location of the directive and its relation to other directives (for example, whether it is nested within another directive, or whether it encloses another directive) determines the effective scope of the option.

You do not have to specify "yes" when you write an option. For example, the option to turn on inlining is:

```
   opt(inline, yes)
```

However, you could write the same option this way:

```
   opt(inline)
```

This is true even when the default setting of the option is "no". For example, if you do not specify the option for run-time type identification, gen(rtti), the default is gen(rtti,no). However, if you specify gen(rtti), it is evaluated as gen(rtti,yes).

4.2 Applying an option to a limited number of sources

We will add the macros(global) option to the configuration file for spp.exe, but will make it apply only to certain source files (those containing macros).

```
target "spp.exe"
{
   option macros(global,yes)
   {
      source "Tools\schema\CommandParser.cpp"
      source "Tools\schema\CommandParser.hpp"
      source "Tools\schema\Preprocessor.cpp"
      source "Tools\schema\Preprocessor.hpp"
      //This is a C++ file, to be inlined:
      source type ("cpp") "iaset.inl"
      //This is a C file, to be processed as C++:
      source type ("cpp") "iisetavl.c"
      //This file does not exist yet. We will create it later:
      //source "SppParser.cpp"
   }
   // These two files do not exist yet. We will create them later.
   //source "SppParser.hpp"
   //source "SppScanner.cpp"
}
```

In this configuration, macros defined in any of the first seven source files will be visible to all the source files in the project. Macros in the remaining source files, SppParser.hpp and SppScanner.cpp, will be visible only within those files.

4.3 Options with project-wide scope

Some options must be set to apply to all the source files in the project. To do this, set the option directive at the outermost scope (do not enclose it in any other directive). An explanation of the project-wide options follows the configuration file.

```
option link(linkWithMultiThreadLib,yes), link(linkWithSharedLib,yes),
incl(searchpath, "."), incl(searchPath, "x:\\src_dir\\5.2\\cmvc\\src"),
define(CICS_W32)
{
   target "spp.exe"
   {
      option macros(global,yes)
      {
         source "Tools\schema\CommandParser.cpp"
         source "Tools\schema\CommandParser.hpp"
         source "Tools\schema\Preprocessor.cpp"
         source "Tools\schema\Preprocessor.hpp"
         //This is a C++ file, to be inlined:
         source type ("cpp") "iaset.inl"
         //This is a template C++ file:
         source type ("cpp") "iisetavl.c"
         //This file does not exist yet. We will create it later:
         //source "SppParser.cpp"
      }
      // These two files do not exist yet. We will create them later.
      //source "SppParser.hpp"
```

```
        //source "SppScanner.cpp"
    }
}
```

- *link(linkWithMultiThreadLib,yes)* links the target using the multithread libraries.
- *link(linkWithSharedLib,yes)* links the target with the shared libraries.
- *incl(searchpath, ".")* adds the current directory to the default search path.
- *define(CICS_W32)* defines the user macro CICS_W32.

4.4 Options in a makefile

Here is the makefile that was first shown in Chapter 3, “An introduction to configuration files” on page 9, with a few additions to show how the options were set:

```
.\NT\bin\spp.exe:
.\NT\obj\Tools\Schema\Preprocessor.obj \
.\NT\obj\Tools\Schema\SppParser.obj \
.\NT\obj\Tools\Schema\SppScanner.obj \
.\NT\obj\Tools\Schema\CommandProcessor.obj
@echo Linking  .\NT\bin\spp.exe
    # here is where link options are set:
icc @<< -Q+ -Tdp -Gm+ -Gd+ -Ft- /B"/NOE" /qautoimported \
/Fe.\NT\bin\spp.exe .\NT\obj\Tools\Schema\Preprocessor.obj
.\NT\obj\Tools\Schema\SppParser.obj
.\NT\obj\Tools\Schema\SppScanner.obj
.\NT\obj\Tools\Schema\CommandProcessor.obj
.
.
.\NT\obj\Tools\Schema\Preprocessor.obj:  \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools\Schema\Preprocessor.cpp \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools\Schema\Build\NT\unistd.h \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools/Schema/Preprocessor.hpp \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools/Schema/CommandProcessor.hpp
    # the compiler is called, and options passed for
    # Preprocessor.obj:
icc @<< -c -D__WINDOWS__ \
-DCICS_W32 -DFwkCLIENT -Gd+ -Q+ -Ft- -Gm+ /qautoimported \
-Ge+ -Ix:\src_dir\5.2\cmvc\src\Tools\Schema\Build\NT \
-I x:\src_dir\5.2\cmvc\src \
-I x:\src_dir\5.2\cmvc\src\Tools\Schema \
-I.\NT\obj\Tools\Schema\Schema

/Fo.\NT\obj\Tools\Schema\Preprocessor.obj \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools\Schema\Preprocessor.cpp
.
.
.\NT\obj\Tools\Schema\SppParser.obj:  \
.\NT\obj\Tools\Schema\SppParser.hpp
.\NT\obj\Tools\Schema\sppParser.cpp
# the compiler is called, and options passed for
# SppParser.obj:
icc -c -D__WINDOWS__ \
-DCICS_W32 -DFwkCLIENT -Gd+ -Q+ -Ft- -Gm+ /qautoimported \
-Ge+ -Ix:\src_dir\5.2\cmvc\src\Tools\Schema\Build\NT \
-I x:\src_dir\5.2\cmvc\src \
-I x:\src_dir\5.2\cmvc\src\Tools\Schema \
-I.\NT\obj\Tools\Schema\Schema
-Fo .NT\obj\Tools\Schema\sppParser.obj $(VOY_OBJ_DIR)\sppParser.cpp
.
.
.\NT\obj\Tools\Schema\SppScanner.obj:  \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\SppParser.hpp
.\NT\obj\Tools\Schema\SppScanner.cpp
icc -c -D__WINDOWS__ \
-DCICS_W32 -DFwkCLIENT -Gd+ -Q+ -Ft- -Gm+ /qautoimported \
-Ge+ -Ix:\src_dir\5.2\cmvc\src\Tools\Schema\Build\NT \
-I x:\src_dir\5.2\cmvc\src \
-I x:\src_dir\5.2\cmvc\src\Tools\Schema \
-I.\NT\obj\Tools\Schema\Schema
-DYY_READ_BUF_SIZE=1 -DYY_BUF_SIZE=16384\ -Fo$(VOY_OBJ_DIR)\SppScanner.obj
.\NT\obj\Tools\Schema\SppScanner.cpp

.\NT\obj\Tools\Schema\CommandProcessor.obj:  \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools\Schema\CommandProcessor.cpp\
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools/Schema/CommandProcessor.hpp
icc @<< -c -D__WINDOWS__ \
-DCICS_W32 -DFwkCLIENT -Gd+ -Q+ -Ft- -Gm+ /qautoimported \
-Ge+ -Ix:\src_dir\5.2\cmvc\src\Tools\Schema\Build\NT \
-I x:\src_dir\5.2\cmvc\src \
-I x:\src_dir\5.2\cmvc\src\Tools\Schema \
-I.\NT\obj\Tools\Schema\Schema
/Fo.\NT\obj\Tools\Schema\CommandProcessor.obj \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools\Schema\CommandProcessor.cpp
.
.
# cleaning up the build environment:
del /Q /F .\NT\obj\Tools\Schema\Preprocessor.obj
del /Q /F .\NT\obj\Tools\Schema\SppParser.obj
del /Q /F .\NT\obj\Tools\Schema\SppScanner.obj
del /Q /F .\NT\obj\Tools\Schema\CommandProcessor.obj
```

You will notice a few more differences between this file and the equivalent configuration file listed in 4.3, “Options with project-wide scope” on page 16:

- In the makefile, the call to the compiler is made explicitly, and is repeated for each object file that must be compiled.
- The options must be repeated with every call to the compiler.
- The makefile uses one- or two-character options, while the configuration file uses English-like names.

4.5 How the makefile options are translated

Table 2 shows how options from the makefile appear in the configuration file.

Table 2. How options from the makefile appear in the configuration file

Makefile option	Equivalent in configuration file
-DCICS_W32	define(CICS_W32) (defines the user macro CICS_W32)
-DFwkCLIENT	define(FwkCLIENT) (defines the user macro FwkCLIENT)
-I x:\src_dir\5.2\cmvc\src (defines a path to only search for #include files)	incl(searchPath, "x:\\src_dir\\5.2\\cmvc\\src") (defines a path to search for all source files, not just #include files)
-Gd+	link(linkWithSharedLib,yes)
-Q+	Not needed : this option suppresses display of the compiler logo
-Ft-	Not needed: this option controls and directs files for template resolution
-Gm+	link(linkWithMultiThreadLib, yes)
-Ge+	Not needed: this option identifies the output as an executable, and is replaced by the target directive

4.6 Option types

The available option types are:

- gen, or code generation options
- link, or linking options
- lang, or language options
- Miscellaneous options (all others). The miscellaneous options include these types:
  - alloc
  - debug
  - define/undefine
  - include
  - info
  - macros
  - misc
  - report

4.7 Common options you might need

You may find some options are frequently required:

Language options

If you use bool, true, or false in your code, you will have to set the lang(nokeyword, "bool"), lang(nokeyword, "true"), and lang(nokeyword, "false") options. This is especially likely if you make use of IBM Open Class.

Link options

link(linkWithMultiThreadLib) and link(linkWithSharedLib), as used in the example above, are needed if you wish to link with the multithread and shared libraries.

Miscellaneous options

incl(searchpath, "*pathname*") is needed if you will be using any files located in directories that are not in the path set by your environment.

define("*macroname*", *value*) can be used to define macros without having to modify sources.

Macro options

We used macros(global, yes) to address compilation errors like the following:

```
CPPC0274E:The name lookup for "function_name" did not find a declaration.
```

As a general rule, you will find it is necessary to apply this option to all of your header files (.h, .hpp, system header files such as iostream, and so on). It may also be necessary in some cases where you have defined macros or functions in .cpp files.

4.8 Other useful references

For tables mapping all options from previous versions of VisualAge C++ to the new options, see the following pages under Build References in the online help:

- Compile Options from Earlier Versions of VisualAge C++ (OS/2, Windows)
- Link Options from Earlier Versions of VisualAge C++ (OS/2, Windows)
- Compile Options from Earlier Versions of VisualAge C++ (AIX) (this includes both compile options and link options)

Chapter 5. Promoting included files to the configuration file

In the previous two chapters, we created a basic configuration file consisting of a single target and several source files. We then applied options to some of the source files to make macros visible, and to resolve compilation errors where function or macro declarations could not be found.

Before we go on to further develop this configuration however, we will begin to optimize it.

5.1 Why optimize so early?

It may seem premature to begin optimizing at this stage: we have very few source files, and the project seems very simple. However, the greatest improvements in build time over the course of the entire project will come from optimizing early, and optimizing continously as you add more source files.

5.2 First steps in optimizing

The most important change you can make to optimize your configuration is to *promote* included source files. This means that you will find sources that are currently included by other sources, but not listed in the configuration, and make them a part of the configuration directly. Sources that are listed with a source directive are called primary sources. Included sources are called secondary sources.

When you launch a rebuild, and the compiler finds a source that is listed in the configuration, it will process it only once. If no changes are found, it will not have to be recompiled at all. When a file name is not found in the configuration file, however, it will be processed every time it is found in any other source.

There are other things you can do to optimize your configuration, but we will start with this step as it has the greatest impact on the efficiency of your builds.

5.3 What to promote?

To use the most typical example of files that are good candidates for promotion, here are the first few lines of the file SppParser.cpp:

```
   #include <stdio.h>
   #include <istring.hpp>
   #include <iostream.h>
   #define YYDEBUG 1
   #define YYSTYPE IString
   //...etcetera...
```

We will promote the three header files to the configuration as follows:

```
option link(linkWithMultiThreadLib,yes), link(linkWithSharedLib,yes),
incl(searchpath, "."), incl(searchPath, "x:\\src_dir\\5.2\\cmvc\\src"),
define(CICS_W32)
{
   target "spp.exe"
   {
      option macros(global,yes)
      {
         //here are the three new primary sources:
         source "stdio.h"
         source "istring.hpp"
         source "iostream.h"
         source "Tools\schema\CommandParser.cpp"
         source "Tools\schema\CommandParser.hpp"
         source "Tools\schema\Preprocessor.cpp"
         source "Tools\schema\Preprocessor.hpp"
         //This is a C++ file, to be inlined:
         source type ("cpp") "iaset.inl"
         //This is a template C++ file:
         source type ("cpp") "iisetavl.c"
         //This file does not exist yet. We will create it later:
         //source "SppParser.cpp"
      }
      // These two files do not exist yet. We will create them later.
      //source "SppParser.hpp"
      //source "SppScanner.cpp"
   }
}
```

The files were added within the macros(global) option because they are header files, and contain declarations and definitions that must be visible to many files.

5.4 Is that all?

No. There are other things you can do to optimize your configuration file. You should also:

- Properly identify sources that should have the macros(global) option applied to them, as early as possible.
- Group source files for manageability and readability. This is explained in Chapter 12.6, “Organizing libFramework.icc by grouping options” on page 65.
- Group options, in cases where the same options are used in several parts of a project. This is demonstrated in Chapter 12.6, “Organizing libFramework.icc by grouping options” on page 65. It is also useful in cases where you have sets of options that may change depending on conditions such as the target operating system. An example of using option groups this way is provided in 6.4, “Variables make conditional processing easier” on page 27.

5.5 If you are migrating

In our example, we promoted three header files to the configuration at once. When you are migrating a project from an earlier version of VisualAge C++, it may be wiser to start by promoting one source or header file at a time. Start with the source or header file that has the most include directives; build; and keep rebuilding as you add additional sources.

5.6 Other useful references

For more details on how to optimize your configuration, read the Migration Guide, available in PDF form, at the following Web site:

```
http://www.software.ibm.com/ad/visualage_c++/downloads.html
```

Chapter 6. Variables

In Chapter 4, “Adding options”, a basic configuration file was developed, containing a target, options, and sources. In real life, for most makefiles, you will probably find that you need more than these to make your configuration file readable and easy to follow and maintain. Often, you need to set variables of your own to do simple things like replace long pathnames, or modify environment variables. This chapter will describe how to do that.

6.1 User-defined variables in configuration files

Variables in configuration files work in the same way variables in makefiles do: they act as a replacement for a string of text.

For example, if the source files we wanted to use were not located in the current working directory, we could use variables to represent the paths. Here is how the example from the previous chapter would look if we had to fully qualify the source and target locations, and wanted to keep the output of the build in a separate directory from the source:

```
option link(linkWithMultiThreadLib,yes), link(linkWithSharedLib,yes),
incl(searchpath, "."), incl(searchPath, "x:\\src_dir\\5.2\\cmvc\\src"),
define(CICS_W32)
{
   target "C:\project\working\code\output\spp.exe"
   {
      option macros(global,yes)
      {
         source "stdio.h"
         source "istring.hpp"
         source "iostream.h"
         source "Tools\schema\CommandParser.cpp"
         source "Tools\schema\CommandParser.hpp"
         source "Tools\schema\Preprocessor.cpp"
         source "Tools\schema\Preprocessor.hpp"
         source type ("cpp") "iaset.inl"
         source type ("cpp") "iisetavl.c"
         //This file does not exist yet. We will create it later:
         //source "C:\project\working\code\src\SppParser.cpp"
      }
      //These two files do not exist yet. We will create them later:
      //source "C:\project\working\code\src\SppParser.hpp"
      //source "C:\project\working\code\src\SppScanner.cpp"
   }
}
```

While this is adequately readable in this small example, this arrangement would quickly become cumbersome if applied to a project with 100 or more files, and would become difficult to maintain as working directories change (for example, if you use the same configuration to build the project on different platforms or machines, and need to specify new working directories).

We will simplify this by defining three simple variables: SRC_DIR, OUTPUT_DIR and TOOLS, described in Table 3.

Table 3. Assigning a value to three variables used for directory paths

Variable name	Variable value
SRC_DIR	c:\project\working\code\src\
OUTPUT_DIR	c:\project\working\code\output\
TOOLS	Tools\Schema\

The syntax for defining a variable name uses the '=' assignment directive as shown in the configuration file in 6.3, "Example of a configuration file using variables" on page 26.

6.2 Rules for using variables

- While compilation of your source files in a VisualAge C++ project is incremental and your declarations do not have to be ordered, the directives and variables in a configuration file are interpreted in the order that they appear. For this reason, you must define variables before you use them.
- To call environment variables that you have set outside the configuration file, prefix the variable name with a '$', for example:

```
$INCLUDE
```

- Variables can define strings or values.
- To define an empty string, use null, for example:

```
test_string=""
```

6.3 Example of a configuration file using variables

When the paths are replaced with variable names, our example now looks like this:

```
SRC_DIR='c:\project\working\code\src\'
OUTPUT_DIR='c:\project\working\code\output\'
TOOLS="Tools\\schema\\"

option link(linkWithMultiThreadLib,yes), link(linkWithSharedLib,yes),
incl(searchpath, "."), define(CICS_W32)
{
   target OUTPUT_DIR "spp.exe"
   {
      option macros(global,yes)
      {
         source "stdio.h"
         source "istring.hpp"
         source "iostream.h"
         source TOOLS "CommandParser.cpp"
         source TOOLS "CommandParser.hpp"
         source TOOLS "Preprocessor.cpp"
         source TOOLS "Preprocessor.hpp"
         source type ("cpp") "iaset.inl"
         source type ("cpp") "iisetavl.c"
```

```
            //This file does not exist yet. We will create it later:
            //source SRC_DIR "SppParser.cpp"
         }
         //These two files do not exist yet. We will create them later:
         //source SRC_DIR "SppParser.hpp"
         //source SRC_DIR "SppScanner.cpp"
      }
   }
```

You will notice that the first two variables SRC_DIR and OUTPUT_DIR use a single quotation mark to enclose the string value whereas the last variable TOOLS uses a double quotation mark. The difference is that the single quotation mark treats the enclosed characters as literals, whereas the double quotation mark will interpret the enclosed characters according to C/C++ programming rules. For example, the backslash character will be treated as an escape character.

6.4 Variables make conditional processing easier

Another example of using variables in a configuration file can be found in the sample project **dllexp**, in the **comp\samples** subdirectory of the main installation directory for VisualAge C++.

This sample uses variables to set up conditional processing for using the same configuration on different platforms, and uses some of the builtin VisualAge C++ macros to do so. It also demonstrates using grouped options.

The contents of the configuration file are as follows:

```
if $__TOS_AIX__
   {
   TARG = "dllexp"
   LIB = "share.a"
    }
else
    {
   TARG = "dllexp.exe"
   LIB = "share.lib"
    }

if $__TOS_OS2__ | $__TOS_WIN__
   option AdditionalOptions = link(extdictionary, no)
if $__TOS_AIX__
   option AdditionalOptions = link(libPathOut,
"/usr/vacpp/samples/comp/dllexp:/usr/vacpp/lib:/usr/lib:/lib")

option
   AdditionalOptions,
   link(debug, no),
   link(linkwithmultithreadlib, yes),
   link(linkwithsharedlib, yes),
   link(padding, no),
   gen(rtti, yes)
{
   target TARG
   {
      option macros(global)
      {
         source type(cpp) "share.h"
      }
      source "dllexp.cpp"
      source LIB
   }
}
```

6.5 Variables in a makefile

Makefiles use variables in much the same way, so there is not much difference between the variables used in this makefile and those used in the configuration file, except that the ’$’ is not needed to call variables set within the configuration file:

```
# variables are set for the compiler, linker, common options,
# and a pathname:
GCPPC=icc
GLINK=icc
FLAGS=-D__WINDOWS__ \
-DCICS_W32 -D_Export= -DFwkCLIENT -Gd+ -Q+ -Ft- -Gm+ /qautoimported \
-Ge+ -Ix:\src_dir\5.2\cmvc\src\Tools\Schema\Build\NT \
-I x:\src_dir\5.2\cmvc\src \
-I x:\src_dir\5.2\cmvc\src\Tools\Schema \
-I.\NT\obj\Tools\Schema\Schema
OBJ_DIR=.\NT\Tools\Schema
ERASE=del /Q /F
.
.\NT\bin\spp.exe:
$(OBJ_DIR)\Preprocessor.obj \
$(OBJ_DIR)\SppParser.obj \
$(OBJ_DIR)\SppScanner.obj \
$(OBJ_DIR)\CommandProcessor.obj
@echo Linking  .\NT\bin\spp.exe
# here is where link options are set:
$(GLINK) @<< -Q+ -Tdp -Gm+ -Gd+ -Ft- /B"/NOE" /qautoimported \
/Fe.\NT\bin\spp.exe $(OBJ_DIR)\Preprocessor.obj
$(OBJ_DIR)\SppParser.obj
$(OBJ_DIR)\SppScanner.obj
$(OBJ_DIR)\CommandProcessor.obj
.
.
$(OBJ_DIR)\Preprocessor.obj:  \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools\Schema\Preprocessor.cpp \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools\Schema\Build\NT\unistd.h \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools/Schema/Preprocessor.hpp \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools/Schema/CommandProcessor.hpp

icc @<< -c $(FLAGS)
/Fo.$(OBJ_DIR)\Preprocessor.obj \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools\Schema\Preprocessor.cpp
.
.
$(OBJ_DIR)\SppParser.obj:  \
$(OBJ_DIR)\SppParser.hpp
$(OBJ_DIR)\sppParser.cpp
```

```
icc -c $(FLAGS) -Fo $(OBJ_DIR)\sppParser.obj $(OBJ_DIR)\sppParser.cpp
.
.
$(OBJ_DIR)\SppScanner.obj:  \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\SppParser.hpp
$(OBJ_DIR)\SppScanner.cpp
icc -c $(FLAGS) -DYY_READ_BUF_SIZE=1 -DYY_BUF_SIZE=16384\ -Fo
$(OBJ_DIR)\SppScanner.obj $(OBJ_DIR)\SppScanner.cpp

$(OBJ_DIR)\CommandProcessor.obj:  \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools\Schema\CommandProcessor.cpp\
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools/Schema/CommandProcessor.hpp
icc @<< -c $(FLAGS) /Fo $(OBJ_DIR)\CommandProcessor.obj \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools\Schema\CommandProcessor.cpp
.
.
$(ERASE) .\NT\obj\Tools\Schema\Preprocessor.obj
$(ERASE) .\NT\obj\Tools\Schema\SppParser.obj
$(ERASE) .\NT\obj\Tools\Schema\SppScanner.obj
$(ERASE) .\NT\obj\Tools\Schema\CommandProcessor.obj
```

6.6 Other useful references

VisualAge C++ predefines certain environment variables and macros to describe operating systems and product level, which can be used to set up conditional processing, for example:

- __TOS_AIX__ (’TOS’ stands for "target operating system")
- __TOS_WIN__
- __TOS_OS2__
- __IBMCPP__ (by default, this is set to __IBMCPP__=400, to represent Version 4.0)
- __ICC_DIR__ (the path of the current configuration file—.icc file)

These and others are described in the pages "Predefined Build Environment Variables" and "VisualAge C++ Predefined Macro Names", in the online help.

Chapter 7. Running external tools

In the previous chapters, we created a configuration to build a target, spp.exe. However, some of the source files that are needed must be generated with external tools. In this chapter we will add directives to the configuration to start these tools.

7.1 The run directive

To launch any processes external to the compiler, use a run directive. The run directive takes the following form:

run [before|after|cleanup] [sources (*source_file_list*)] [targets (*target_file_list*)] *commands*

- The flags **before**, **after**, and **cleanup** are optional. You only need to use these if you want to ensure that the process being launched is started at a specific stage in compilation ("before" is run before compilation; "after" is run after compiling and linking; "cleanup" is run only when you select "Clean" from the Project Workbook menu to delete the codestore and clean up the build environment).
- The **sources** and **targets** parameters are also optional. These are not passed to the external tool itself. They are used by the incremental compiler to determine dependencies between multiple run directives and to check whether the run directive needs to be repeated on every rebuild.
- The **commands** are those commands you need to be executed by the external command line interpreter. The commands can include executing DOS commands like del and copy, or they can be external tools with options passed to the tools. The commands must be enclosed in quotation marks. You can issue a series of commands to be processed in a sequence by enclosing each command in quotation marks and separating the commands with commas.

7.2 Running tools before compiling and linking

In this example, we need to run two tools, called bison and flex, with different sets of options, to take source files of types .y and .l, and process them into .cpp and .hpp files to build spp.exe.

Since it is imperative that these processes be completed before the compiler starts to look for the source files for spp.exe, we will use the **before** flag. We will also specify sources and targets:

```
run before sources ("Tools/Schema/SppParser.y") targets "("SppParser.cpp",
"SppParser.hpp")

"bison -d -l -t -bSppParser Tools/Schema/SppaParser.y",
"copy SppParser.tab.c SppParser.cpp",
"copy SppParser.tab.h SppParser.hpp"
"del SppParser.tab.c"
"del SppParser.tab.h"

run before sources ("Tools/Schema/SppScanner.l") targets ("SppScanner.cpp")

"flex -t Tools/Schema/SppScanner.l > SppScanner.cpp"
```

The following rules apply to the run directive:

- You can have several run directives in sequence. Run directives are executed according to the order of the dependencies between them, which are determined by the parameters of the sources and targets flags.
- Command strings are executed in the order they appear.
- The command strings will continue to be read in until the next configuration file directive is encountered. You can issue as many commands as you need to.
- If the source files for the external tool are not located in the project working directory (usually the same directory as your configuration file), you must qualify the file paths.
- You can list as many sources as you need to, but they must appear within the (and) of a single sources parameter. You cannot repeat the source parameter in the same way that you can repeat the source directive.

While the directive can appear almost anywhere in the configuration file, we have enclosed it in the target directive so that there will be no confusion for future users of our configuration file regarding which part of the project this tool stage is associated with.

This is how the configuration file for building spp.exe looks now (note that we have removed the comments (//) from the three source files that previously could not be found):

```
option link(linkWithMultiThreadLib,yes), link(linkWithSharedLib,yes),
incl(searchpath, "."), incl(searchPath, "x:\\src_dir\\5.2\\cmvc\\src"),
define(CICS_W32)
{
   target "spp.exe"
   {
      run before
      sources ("Tools/Schema/SppParser.y")
      targets ("SppParser.cpp",SppParser.hpp")

      "bison -d -l -t -bSppParser Tools/Schema/SppaParser.y",
      "copy SppParser.tab.c SppParser.cpp",
      "copy SppParser.tab.h SppParser.hpp"
      "del SppParser.tab.c"
      "del SppParser.tab.h"

      run before
      sources ("Tools/Schema/SppScanner.l")
      targets ("SppScanner.cpp")

      "flex -t Tools/Schema/SppScanner.l > SppScanner.cpp"

      option macros(global,yes)
      {
         source type('cpp') 'stdio.h'
         source 'istring.hpp'
         source type('cpp') 'iostream.h'
         source "Tools\schema\CommandParser.cpp"
         source "Tools\schema\CommandParser.hpp"
         source "Tools\schema\Preprocessor.cpp"
         source "Tools\schema\Preprocessor.hpp"
         source type('cpp') 'iisetavl.c'
         source type('cpp') 'iset.inl'
         source type('cpp') 'iset.h'
         source 'SppParser.cpp'
      }
      source 'SppParser.hpp'
      source 'SppScanner.cpp'
   }
}
```

In this file, our decision to place the run directive within the scope—inside the { and }—of the target directive was based purely on our organizational style: we wanted to be able to easily determine in the future, which target this step was needed for. As the project grows, if the run directive is placed at the top of the configuration file, it may not be as easy to figure out where in the process the external tool was needed, especially if there is eventually more than one target in the same configuration.

Although the directive would still be processed correctly if placed outside the target directive, we recommend that you place run directives belonging to a specific target inside that target.

7.3 Tool commands in a makefile

Here is the makefile equivalent to the configuration file we have developed so far. Changes made since the last chapter are marked with comments (#):

```
GCPPC=icc
GLINK=icc
FLAGS=-D__WINDOWS__ \
-DCICS_W32 -D_Export= -DFwkCLIENT -Gd+ -Q+ -Ft- -Gm+ /qautoimported \
-Ge+ -Ix:\src_dir\5.2\cmvc\src\Tools\Schema\Build\NT \
-I x:\src_dir\5.2\cmvc\src \
-I x:\src_dir\5.2\cmvc\src\Tools\Schema \
-I.\NT\obj\Tools\Schema\Schema
OBJ_DIR=.\NT\Tools\Schema
ERASE=del /Q /F
.
.\NT\bin\spp.exe:
$(OBJ_DIR)\Preprocessor.obj \
$(OBJ_DIR)\SppParser.obj \
$(OBJ_DIR)\SppScanner.obj \
$(OBJ_DIR)\CommandProcessor.obj
@echo Linking  .\NT\bin\spp.exe
$(GLINK) @<< -Q+ -Tdp -Gm+ -Gd+ -Ft- /B"/NOE" /qautoimported \
/Fe.\NT\bin\spp.exe $(OBJ_DIR)\Preprocessor.obj
$(OBJ_DIR)\SppParser.obj
$(OBJ_DIR)\SppScanner.obj
$(OBJ_DIR)\CommandProcessor.obj
.
.
$(OBJ_DIR)\Preprocessor.obj:  \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools\Schema\Preprocessor.cpp \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools\Schema\Build\NT\unistd.h \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools/Schema/Preprocessor.hpp \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools/Schema/CommandProcessor.hpp
icc @<< -c $(FLAGS)
/Fo.$(OBJ_DIR)\Preprocessor.obj \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools\Schema\Preprocessor.cpp
.
.
$(OBJ_DIR)\SppParser.obj:\
# this file is required as input to the 'bison' tool, in order
# to produce the C++ files:
x:\src_dir\5.2\cmvc\src\Tools\Schema\SppParser.y
# these are the commands to run bison and create .cpp and .hpp files:
bison -d -l -t -b SppParser Tools/Schema/SppaParser.y
copy SppParser.tab.c SppParser.cpp
copy SppParser.tab.h SppParser.hpp
del SppParser.tab.c
del SppParser.tab.h

$(OBJ_DIR)\SppParser.hpp
$(OBJ_DIR)\sppParser.cpp

icc -c $(FLAGS) -Fo $(OBJ_DIR)\sppParser.obj $(OBJ_DIR)\sppParser.cpp
.
.
$(OBJ_DIR)\SppScanner.obj:  \
# this file is required as input to the 'flex' tool, to create
# SppScanner.cpp:
x:\src_dir\5.2\cmvc\src\Tools\Schema\SppScanner.l
# these are the commands to run flex and generate SppScanner.cpp:
flex -t Tools/Schema/SppScanner.l > $(OBJ_DIR)SppScanner.cpp

H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\SppParser.hpp
$(OBJ_DIR)\SppScanner.cpp
icc -c $(FLAGS) -DYY_READ_BUF_SIZE=1 -DYY_BUF_SIZE=16384\ -Fo
$(OBJ_DIR)\SppScanner.obj $(OBJ_DIR)\SppScanner.cpp

$(OBJ_DIR)\CommandProcessor.obj:  \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools\Schema\CommandProcessor.cpp\
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools/Schema/CommandProcessor.hpp
icc @<< -c $(FLAGS) /Fo $(OBJ_DIR)\CommandProcessor.obj \
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\Tools\Schema\CommandProcessor.cpp
.
.
$(ERASE) .\NT\obj\Tools\Schema\Preprocessor.obj
$(ERASE) .\NT\obj\Tools\Schema\SppParser.obj
$(ERASE) .\NT\obj\Tools\Schema\SppScanner.obj
$(ERASE) .\NT\obj\Tools\Schema\CommandProcessor.obj
```

7.4 Other useful references

For the complete syntax of the run directive, see the online help page "Run Configuration File Directive".

See "Run Command Line Programs During Builds" in the online help for more information on launching tools other than the compiler in your build.

Chapter 8. Building a shared (dynamic link) library

In this chapter we take a look at building a shared library on AIX named libFramework.so or the equivalent Dynamic Link Library (DLL) on Windows NT named libFramework.dll. Shared libraries contain compiled source code which can be indirectly linked (referenced) to other shared libraries or executables. Building a shared library involves compiling and linking.

This chapter uses the terms shared library and DLL interchangeably to address both AIX and Intel platforms.

8.1 Advantages of shared libraries

There are several performance advantages of building reusable source code into a shared library:

- The source code is only compiled and linked once.
- Run-time memory consumption is reduced, as the source code is linked into only one library.
- Disk storage use is reduced, as the source code is compiled and linked into only one library.

The impact of these advantages is even greater if the DLL is referenced by many other DLLs or executables.

In addition, using DLLs can make it easier for a developer to work within a project team. Partitioning a few large libraries or executables into many small libraries assists project team development. Developers can work independently of each other by working on different small libraries. When the interfaces between the libraries need to be worked on and tested, then integration can be performed.

8.2 Configuration files simplify building dynamic link libraries

Here we focus on showing you how configuration files simplify the following aspects of building a DLL:

- There is no need to use a compiler switch to indicate that the source code is being compiled into a DLL versus an executable.
- There is no need to know how to produce import libraries for Intel platforms.
- There is no need to invoke the linker as a separate process.
- There is no need to invoke tools to export all symbols.

Where a typical batch compiler requires you to know how to manipulate the tools to get the right switches and sequence of tool invocations, VisualAge C++ only requires you to know what you want to perform, which is to build a DLL.

8.3 Linking against other dynamic link libraries

In 3.2, “Sources” on page 9, you learned to compile source code by using the source directive in a configuration file. Similarly, to link against another library, you use the source directive, and name the library you want to link against as the source file. It is as simple as that.

8.4 About the example

The example shows how one configuration file can be used to build either a shared library on AIX or a DLL on Windows NT.

Before any source of libFramework.dll can be compiled, the schema tool needs to be invoked with certain .s files as input. To accomplish this in the configuration file, the run directive is used to specify:

- When the schema tool is to be run (before the compilation process).
- Which .s file will be input (source) to the schema tool.
- The outputs (targets) created by the schema tool that are used as input to the compilation process. These are .cpp and .hpp files containing C++ source code.

This configuration also makes use of user-defined variables to replace pathnames, and two named option groups, ProjectOptions and PlatformOptions. The link(libSearchPath, *path*) option specifies a directory where library files can be found. This will allow the linker to find the two import libraries on an Intel platform.

8.5 The libFramework.icc configuration file

An excerpt from the libFramework.icc configuration file below shows you how you can build a DLL, link against other libraries and run an external tool. It shows two import libraries, cclicw32.lib and cclcpw32.lib, which are linked to libFramework.dll.

```
//
// This option group is used to set options that must have project-
// wide scope:
//
option ProjectOptions = gen(rtti, "all")
{
   //
   // These variables replace filepaths for certain sources,
   // and make future maintenance easier (the path will only have
   // to be changed in one location):
   //

   if $__TOS_AIX__
   {
      DIRSEP = "/"
      PLATFORM = "AIX"
      BASE_DIR = "/home/common/ActiveOrder/fwk3.5/3.5.2"
      TARGET = "libFramework.so"
      LIBFWKCORE = "libFwkCore.so"
      LIBCICS = "libcicsicc.a"
      option PlatformOptions = define("FwkSERVER"),
         link(libSearchPath, "/usr/lpp/cics/lib")
   }
   else
   {
      DIRSEP = "\\"
      PLATFORM = "NT"
      BASE_DIR = "h:\\common\\ActiveOrder\\fwk3.5\\3.5.2"
      TARGET = "libFramework.dll"
      LIBFWKCORE = "libFwkCore.lib"
      LIBCICS = "cclicw32.lib, cclcpw32.lib"
      option PlatformOptions = define("FwkCLIENT"),
         define("_X86_"), define("CICS_W32"),
         link(subsystem, "windows", 4, 0),
         link(libSearchPath, "c:\\cicscli\\lib")
   }
   SRC_DIR = BASE_DIR DIRSEP "cmvc" DIRSEP "src"
   GEN_DIR = BASE_DIR DIRSEP "gen"
   OBJ_DIR = GEN_DIR DIRSEP PLATFORM DIRSEP "obj_dbg"
   SCHEMA_BIN    = SRC_DIR DIRSEP "schema"
   SCHEMAGEN_DIR = OBJ_DIR DIRSEP "ProdSoft" DIRSEP "Framework"
                   DIRSEP "Schema"

   option link(linkwithmultithreadlib), link(linkwithsharedlib),
      lang(nokeyword, "bool"), lang(nokeyword, "true"),
      lang(nokeyword, "false"), define("IC_LANG_BOOL", 0),
      define("IVB_IMPORT", "_Import"), link(exportAll, yes),
      incl(searchpath, "."),
      incl(searchpath, OBJ_DIR "/ProdSoft/Framework/Schema"),
      PlatformOptions
   {
      target TARGET
      {
         source
            LIBCICS, LIBFWKCORE

         run before
            sources("ProdSoft/Framework/Model/FwkMdlMapAttribute.s")
            targets("FwkMdlMapAttribute.cpp","FwkMdlMapAttribute.hpp",
                    "FwkMdlMapAttributeAttrib.cpp",
                    "FwkMdlMapAttributeAttrib.hpp")
            SCHEMA_BIN " -o" SCHEMAGEN_DIR " -s "
               SRC_DIR DIRSEP "ProdSoft" DIRSEP "Framework"
               DIRSEP "Model" DIRSEP "FwkMdlMapAttribute.s"
         //
         // Source directives.
         //
         option macros('global', 'yes')
         {
            source "FwkMdlMapAttributeAttrib.hpp"
         }
         source "FwkMdlMapAttribute.cpp","FwkMdlMapAttribute.hpp",
            "FwkMdlMapAttributeAttrib.cpp"}
      }
   }
}
```

Some options you may not have seen before appear in this file:

- define("_X86_") states that the build will occur on an Intel x86 system. Do not use this option if you are building on AIX.
- $__TOS_AIX__ states the target operating system you are building for is AIX.

- link(subsystem, windows,4,0) identifies the subsystem and the minimum required level of the operating system. This option is only needed if you are developing an application for use with Windows.

8.6 How do external tools fit into the incremental build process?

The first time this configuration is built, the tools called in the *run before* directive will be launched, compilation will begin when the tools' targets are completed, and a codestore will be created.

On subsequent (incremental) builds, the *run before* directive will only be carried out if the timestamps of the sources show that they have been changed since the previous build. If they have not changed, the step to run the external tools will be skipped.

Chapter 9. Building an archive (static library)

In this chapter we take a look at building an archive named libUIFwk.a on AIX or a static library named libUIFwk.lib on Windows NT. Static libraries contain compiled source code which can be later linked (copied) into one or more Dynamic Link Libraries (DLL) or executables. Building a static library involves the compilation process but does not involve linking, because the compiled source code is to be later linked into a DLL or executable.

9.1 Static linking versus dynamic linking

The interesting point to observe with building libUIFwk.lib is the amount of memory that is consumed by VisualAge C++ in the compilation process: approximately 380Mb, in the case of our code, which is an excessive amount of memory. The source code in this library happens to be both compilable and linkable. Hence, building it into a DLL reduces the amount of memory used to approximately 80Mb, almost a five-fold reduction.

As much as possible, avoid building static libraries (unless they are small) and instead build the source code into its final target, a DLL or an executable.

9.2 Configuration for building a static library

What follows is the configuration file for building libUIFwk.lib. You will notice the absence of link(...) options, since the building of static libraries does not involve a link phase.

```
option UIFwkProjectOptions = gen(rtti, yes)
{
   if $__TOS_AIX__
   {
      DIRSEP = "/"
      PLATFORM = "AIX"
      BASE_DIR = "/home/common/ActiveOrder/fwk3.5/3.5.2"
      TARGET = "libUIFwk.a"
      option PlatformOptions = define("FwkSERVER")
   }
   else
   {
      DIRSEP = "\\"
      PLATFORM = "NT"
      BASE_DIR = "h:\\common\\ActiveOrder\\fwk3.5\\3.5.2"
      TARGET = "libUIFwk.lib"
      option PlatformOptions = define("FwkCLIENT"),
         define("_X86_"), link(subsystem, "windows", 4, 0)
   }
   SRC_DIR = BASE_DIR DIRSEP "cmvc" DIRSEP "src"
   GEN_DIR = BASE_DIR DIRSEP "gen"
   OBJ_DIR = GEN_DIR DIRSEP PLATFORM DIRSEP "obj_dbg"

   option
      incl(searchpath, "."), lang(nokeyword, "bool"),
      lang(nokeyword, "true"), lang(nokeyword, "false"),
      define("IC_LANG_BOOL", 0), define("IVB_IMPORT", "_Import"),
      incl(searchpath,
         SRC_DIR "/BeckerMohnberg/" PLATFORM "/Chart/include"),
      incl(searchpath, OBJ_DIR "/ProdSoft/Framework/Schema"),
      PlatformOptions
   {
      target TARGET
      {
            //
            // source directive and list of source to compile goes
            // here.
            //
      }
   }
}
```

The gen(rtti) option sets run-time type information. The gen(rtti,yes) setting generates code that supports both the typeid and dynamic_cast operators.

9.3 Changing libUIFwk.icc to build a DLL

To now build libUIFwk.lib as a DLL simply involves:

- Changing the target extension from .lib to .dll.
- Adding the required link(...) options.
- Specifying the other libraries to link to.

The configuration file to accomplish this is shown below.

```
option UIFwkProjectOptions = gen(rtti, yes)
{
   if $__TOS_AIX__
   {
      DIRSEP = "/"
      PLATFORM = "AIX"
      BASE_DIR = "/home/common/ActiveOrder/fwk3.5/3.5.2"
      TARGET = "libUIFwk.so"
      option PlatformOptions = define("FwkSERVER")
      LIBFWK = "libFramework.so"
      LIBOTHER = "bchart.a"
   }
   else
   {
      DIRSEP = "\\"
      PLATFORM = "NT"
      BASE_DIR = "h:\\common\\ActiveOrder\\fwk3.5\\3.5.2"
      TARGET = "libUIFwk.dll"
      option PlatformOptions = define("FwkCLIENT"),
         define("_X86_"), link(subsystem, "windows", 4, 0)
      LIBFWK = "libFramework.lib"
      LIBOTHER = "bchart.lib", "user32.lib", "gdi32.lib",
         "comctl32.lib"
   }
   SRC_DIR = BASE_DIR DIRSEP "cmvc" DIRSEP "src"
   GEN_DIR = BASE_DIR DIRSEP "gen"
   OBJ_DIR = GEN_DIR DIRSEP PLATFORM DIRSEP "obj_dbg"
   SCHEMA_BIN    = SRC_DIR DIRSEP "schema"
   SCHEMAGEN_DIR = OBJ_DIR DIRSEP "ProdSoft" DIRSEP "Framework"
```

```
                DIRSEP "Schema"

   option link(linkwithmultithreadlib), link(linkwithsharedlib),
      incl(searchpath, "."), lang(nokeyword, "bool"),
      lang(nokeyword, "true"), lang(nokeyword, "false"),
      define("IC_LANG_BOOL", 0), define("IVB_IMPORT", "_Import"),
      incl(searchpath,
         SRC_DIR "/BeckerMohnberg/" PLATFORM "/Chart/include"),
      incl(searchpath, OBJ_DIR "/ProdSoft/Framework/Schema"),
      link(libSearchPath,
         SRC_DIR "/BeckerMohnberg/" PLATFORM "/Chart/lib"),
      link(exportAll, yes), PlatformOptions
   {
      target TARGET
      {
      //
      // source directive and list of source to compile goes
      // here.
      //
      // These are the external libraries:
      source LIBFWK, LIBOTHER
      }
   }
}
```

To link with the library bchart.lib, we use the link(libSearchPath, *path*) option, as bchart.lib is not in the current directory, nor in any directory specified in the LIB environment variable.

Chapter 10. Building a resource library

In this chapter we take a look at how you can build a resource library that can contain various resource types such as strings, icons, bitmaps, and menu definitions. We compare a configuration and makefile to show you the difference between building with VisualAge C++ Version 4.0 and older batch compiler technology.

10.1 About the example

The configuration file rSecMan.icc is an example of building resources into a Dynamic Link Library (DLL).

The source files used to build this DLL are of the file extension .rc, which is recognized by VisualAge C++ as a resource source file. An option directive is used to create a named group of options for the resource compiler (IRC).

In this example configuration you will notice:

- A variable is created to replace a long path name.
- There is no explicit call to the resource compiler. Since the C++ compiler recognizes the .rc extension, the resource compiler will be launched in the background.

The equivalent makefile that was previously used to compile the same source is included following this example.

10.2 Configuration file for a resource DLL

The content of the rSecMan.icc file follows:

```
option incl(searchpath, ".")
{
   target "rSecMan.dll"
   {
      option res_rc_options( "-i ProdSoft\\IOCUI\\Resources\\include" )
      {
        ENGLISH_SUBDIR = "ProdSoft\\IOCUI\\Resources\\NT\\LOCALE\\EN_NZ"
        source
          ENGLISH_SUBDIR "\\SecurityProfileInfo.rc",
          ENGLISH_SUBDIR "\\SecurityProfileSetPassword.rc",
          ENGLISH_SUBDIR "\\MaintainSecurityGroups.rc",
          ENGLISH_SUBDIR "\\secManFrame.rc",
          ENGLISH_SUBDIR "\\SecManProductInformation.rc",
          ENGLISH_SUBDIR "\\SecManToolbar.rc",
          ENGLISH_SUBDIR "\\logon.rc",
          ENGLISH_SUBDIR "\\ChangeUserPassword.rc",
          ENGLISH_SUBDIR "\\global.rc",
          ENGLISH_SUBDIR "\\icons.rc"
       }
   }
}
```

The option res_rc_options(*string*) is used to pass commands to the resource compiler.

10.3 Makefile for a resource DLL

The content of the rSecMan.mak file follows:

```
# Predefined symbols
ERASE=del /Q /F
GCPPC=icc
GSLINK=icc

GCPPFLAGS=\
   -Gd+ -Q+ -Ft- -Gm+ /qautoimported -Ge-\
   -I$(SRC_DIR)\ProdSoft\IOCUI\Resources\NT\LOCALE\EN_NZ
GRC = irc -v
PROGRAM = rSecMan
RC_INCLUDE = -i $(SRC_DIR)\ProdSoft\IOCUI\Resources\include
SRC_DIR = H:\common\ActiveOrder\C11U\bld\cmvc\src

all: rSecMan.dll

rSecMan.dll: rSecMan.obj dummy.obj
echo LIBRARY rSecMan >$(PROGRAM).def
@echo Linking rSecMan.dll
$(GSLINK) @<<
/B" /de /nobrowse /pmtype:vio /noe /code:RX /data:RW /dll"
/B" /def"  /B" /nod:$(PROGRAM).lib"
/Fe$(PROGRAM).dll $(PROGRAM).res
dummy.obj
<<
$(GRC) $(PROGRAM).res \$(PROGRAM).dll
$(ERASE) $(PROGRAM).def

rSecMan.obj: \
   $(SRC_DIR)\ProdSoft\IOCUI\Resources\NT\LOCALE\EN_NZ\rSecMan.rc
$(GRC) $(RC_INCLUDE) -r \
         $(SRC_DIR)\ProdSoft\IOCUI\Resources\NT\LOCALE\EN_NZ\rSecMan.rc \
         /fo$(PROGRAM).res

dummy.obj: $(SRC_DIR)\ProdSoft\dummy.cpp
@echo Compiling  \
$(SRC_DIR)\ProdSoft\dummy.cpp
@echo into dummy.obj
$(GCPPC) @<<
-c $(GCPPFLAGS)
/Fodummy.obj
$(SRC_DIR)\ProdSoft\dummy.cpp
<<
```

The file rSecMan.rc has the following content:

```
rcinclude "SecurityProfileInfo.rc"
rcinclude "SecurityProfileSetPassword.rc"
rcinclude "MaintainSecurityGroups.rc"
rcinclude "SecManFrame.rc"
rcinclude "SecManProductInformation.rc"
rcinclude "SecManToolbar.rc"
rcinclude "logon.rc"
rcinclude "ChangeUserPassword.rc"
rcinclude "global.rc"
rcinclude "icons.rc"
```

The file rSecMan.rc is compiled, rather than the included files, to speed the compilation process and to reduce the size of the makefile.

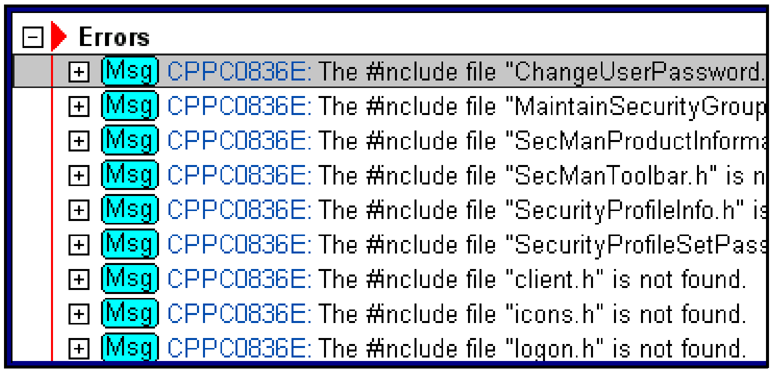
10.4 Differences between the configuration file and makefile

The main differences you will notice between the two types of files are the following:

- The configuration file does not need to state a C++ compiler, linker, and resource compiler (tools) to be used. These are intrinsically known by VisualAge C++.
- The configuration file does not need to show how the various tools are to be executed and in what order.
- The configuration file does not need to build a dummy object file and link it in order to satisfy the linker’s need for at least one object file to be linked.

10.5 Problems with building the resource library

There is a defect in VisualAge C++ Version 4.0 Fixpack 1 for Windows NT that causes the build of a resource library to generate false error messages (CPPC0836E, shown in Figure 4) for missing include files, even though the target was built. Also, the status bar reports that the build was successful and only had warnings. Until a fix is released, the extraneous error messages can ordinarily be ignored when building the resource library on its own. The errors could not be ignored, however, for building our entire small software project from a clean build as discussed in 13.3.3, “Problems with executing our build process” on page 76. That section discusses the workaround you need to perform.



```
Errors
Msg CPPC0836E: The #include file "ChangeUserPassword.
Msg CPPC0836E: The #include file "MaintainSecurityGroup
Msg CPPC0836E: The #include file "SecManProductInform
Msg CPPC0836E: The #include file "SecManToolbar.h" is n
Msg CPPC0836E: The #include file "SecurityProfileInfo.h" is
Msg CPPC0836E: The #include file "SecurityProfileSetPass
Msg CPPC0836E: The #include file "client.h" is not found.
Msg CPPC0836E: The #include file "icons.h" is not found.
Msg CPPC0836E: The #include file "logon.h" is not found.
```

Figure 4. VisualAge C++ defect when building rSecMan.dll

You may encounter another defect if you have more than one resource file and you are compiling them into an executable. For this scenario, if you modify your source code and perform an incremental build, then the build is likely to fail with an error when invoking the resource compiler. The workaround is to build your resources into a DLL as was done in this chapter.

In any case, it is a much better practice to build your resources into a DLL, rather than an executable, for the following reasons:

- It is faster to bind your resources into a small DLL than a medium- to large-sized executable.
- Every time the executable is rebuilt, the resources will need to be rebound to the executable.
- It is possible to replace the resource DLL with another without having to recompile your application. This is needed in the case of multilingual applications.

10.6 Other useful references

For information on using resource files with VisualAge C++, see the following pages in the online Task help:

- Use Resource Source Files as Sources for a Build
- Compile and Bind Resources

Chapter 11. Building two targets in one configuration

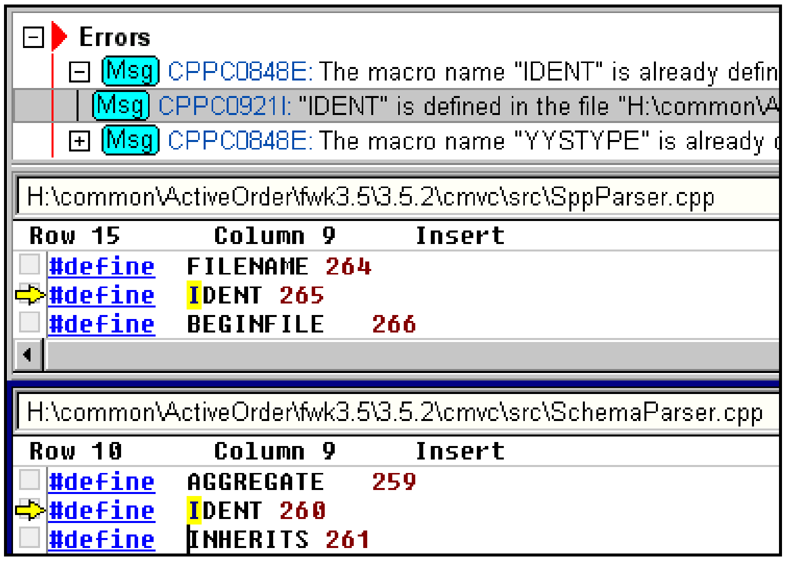
In this chapter we take a look at building more than one target in one configuration file. There can be problems associated with this that will cause you to break up the configuration file into multiple configuration files. This chapter will explore those problems, explain their cause, and show you what you can do to resolve them.

11.1 Building schema and spp in one configuration

In Chapter 7, “Running external tools”, we built a single executable, spp.exe. Now we will try to build two executables in one configuration file by listing both targets, and observe what happens when we try to build. A configuration file with both targets can consist simply of the following lines:

```
include "schema.icc"
include "spp.icc"
```

When this is built, the result is the same as replacing the full text of both files into one configuration. However, doing this results in the error messages shown in Figure 5.



```
Errors
  Msg CPPC0848E: The macro name "IDENT" is already defin
    Msg CPPC0921I: "IDENT" is defined in the file "H:\common\A
  Msg CPPC0848E: The macro name "YYSTYPE" is already d

H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\SppParser.cpp
Row 15      Column 9      Insert
#define  FILENAME 264
#define  IDENT 265
#define  BEGINFILE    266

H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\SchemaParser.cpp
Row 10      Column 9      Insert
#define  AGGREGATE    259
#define  IDENT 260
#define  INHERITS 261
```

Figure 5. Errors produced when the one-definition rule is broken

Both targets define several macros of the same name with different values. This is not permitted within the same codestore. VisualAge C++ enforces the *one-definition rule:* a definition can only occur once per codestore. Hence, in order to build schema and spp, they need to be built separately into separate codestores.

11.2 Keeping in step with the one-definition rule

VisualAge C++ enforces the C++ programming rule to not declare or define the same object across multiple source files with either the same or different declarations or definitions. That is in part why header files were introduced. The standard practice is for a header file to contain declarations and the source file to contain the definitions (implementations) of those objects declared in the header file. And for the same object there should be just one declaration and one definition.

It is possible to build more than one target in the same codestore as long as you abide by the rule to have unique declarations or definitions.

11.3 Breaking the one-definition rule within one target

A codestore permits only one declaration or definition of an object. Earlier in this chapter you saw that building multiple targets from one configuration file can cause the one-definition rule to be broken. This is not the only scenario to break the one-definition rule.

The rule can also be broken with just one target in a configuration file. This typically happens when declarations occur within source files instead of header files. It is a good programming practice to place declarations only within header files to avoid potential scenarios of breaking the one-definition rule.

An example of breaking the one-definition rule within one target is shown below. Here we have two source files, both declaring the same structure, Tree. The two structures have the same name, although their contents are different.

```
//
// Excerpt from file compiler.cpp
//
struct Tree
{
   TreePtr   left;
   TreePtr   right;
   CompilerTool2Proc t; // This is better than a pointer to the
                        // array of compilers since it lets me
                        // add new ones in the future dynamically
};

//
// Excerpt from file extensio.cpp
//
struct Tree
{
   TreePtr        left;
   TreePtr        right;
   Extension2Type t;
};
```

The solution is to rename one of the structures to CompilerTree. Be careful when changing the source code to ensure all uses of the object you are changing are renamed. Otherwise, further compilation errors and/or run-time errors will occur. The renamed structure now becomes:

```
struct CompilerTree
{
   CompilerTreePtr   left;
   CompilerTreePtr   right;
   CompilerTool2Proc t; // This is better than a pointer to the
                        // array of compilers since it lets me
```

```
                              // add new ones in the future dynamically
};
```

Several other lines of code within the same source file had to be changed to refer to structure CompilerTree instead of structure Tree to ensure there were no further compilation errors and/or run-time errors.

11.4 Why is the one-definition rule not enforced by other compilers?

Batch compilers compile source code by executing the compiler with a given source file as input. Hence, a compiler only sees the declarations or definitions within the scope of that source file. When the compiler compiles the next source file, it has already discarded the information from memory of compiling the previous source file. Thus, if a programmer declares or defines the same thing in two source files, the compiler will be none the wiser and will happily compile both source files.

The only way a batch compiler could possibly enforce the one-definition rule would be to compile all source code in one pass. This can be accomplished by writing a meta-source file that includes more than one real source file and compile that one meta-source file instead of each individual real source file. An example of this follows:

```
//
// Excerpt from metasrc.cxx
//
#include "compiler.cpp"
#include "extensio.cpp"
```

11.5 Other useful references

For a complete explanation of the one-definition rule, refer to Section 3.2 of the ISO C++ Standard document.

Part 2. Broader changes to your projects

In the first part of this book, we focused on making the change from configuration files to makefiles. However, the makefile is not the only part of your project that will be impacted by the new method of compilation.

In this section we will look at ways in which the structure and organization of your project can be adjusted to help take advantage of the codestore and the IDE.

Chapter 12. Designing configuration files for project teams

In this chapter we will take a look at how you can design your configuration files when many developers are working in a project team. There are a number of approaches a project team can take. We will present one recommended approach. You will need to consider the following:

- The team structure of your project if there are multiple teams within a project. Also consider the dependencies between teams.
- The granularity of your components (executables and libraries).
- The dependencies between your components.
- The amount of reuse present in your configuration files.

This chapter will show you how carefully designing your configuration files will lead to better management of readability and maintenance.

12.1 Splitting a project into architectural areas

All projects other than trivial ones will have many components (executables, dynamic link libraries, static libraries and resource libraries). In an ideal situation, many developers hungry to write C++ source code will be resourced to develop these components. It is usually the responsibility of the project's architect to work out how a project can be split into components.

A common first split of a project is into three main architectural areas:

- The infrastructure or framework. This normally contains the implementation of the non-functional requirements, and it forms the foundation for the application to be built on top of it.
- The application's user interface. This contains the many window designs, navigation behavior between windows and window validation logic.
- The application business logic. The contents are the implementation of the functional requirements provided by the business users.

A project is usually split this way due to the type of expertise expected from a developer working in each area.

12.1.1 Infrastructure

The expertise required by a developer working in this domain is very specialized. It has no bearing on the direct functional requirements, as it mainly involves in-depth knowledge of software and hardware such as: middleware, communication protocols, third-party libraries, operating system services, hardware platform, development standards and practices, developer tools.

It is quite common for the developers in this area to be very experienced. They can be significantly involved in leading developers in the other teams, as their experience and the nature of their work requires the other components to adhere to the standards and practices that they establish.

12.1.2 User interface

Developers working in this domain should have a lot of skills with good user interface design. Familiarity with developer tools for building user interfaces is very important too. It is usually quite difficult to build event-driven graphical user interfaces in comparison to common procedural/message programming.

12.1.3 Business logic

The expertise required by developers for this domain is less technology-driven than the others. Instead it requires developers to learn more about the business workflow and how their implementation will interact with the user interface and infrastructure.

12.2 Your project's directory structure

Medium- to large-sized projects need to design a directory structure to reflect their expected architecture and component design. For instance, the following is a hypothetical example of part of a project's directory structure:

```
{root}
    | Infrastructure
        | Middleware
        | Comms
    | Application A
        | UserInterface
        | BusinessModel
        | DatabaseAccess
    | Application B
        | UserInterface
        | BusinessModel
        | DatabaseAccess
...etcetera
```

12.2.1 Reasons for having a directory structure

Designing and implementing a directory structure for all of your project's files is an important step to ease the management of medium- to large-sized projects. Some of the reasons why you should carefully plan a directory structure for your project are as follows:

- Using a directory structure for your source and header files is done for the same reason you choose to have directories on your disk drive for your applications: you want to organize your files into areas that make it easy to find a certain file in the future.
- A directory structure will help maintain good build performance as the number of files increases. If you were to have thousands of files in one directory, the build performance would suffer, as the filesystem performance usually degrades under these conditions.
- A directory structure will make it easier to package and transport individual components. It is easier to do this when all the files for one component can be easily found in one directory and its subdirectories. If the files are not isolated, you have to selectively choose each individual file from among all the other project files when you want to build a component.

12.2.2 Avoiding time-consuming directory searches during builds

Once you have decided on a directory structure, you can use the incl(searchpath, *path*) option to tell the compiler where to look for your files. However, it is better to avoid using a long list of incl(searchpath, *path*) options.

When VisualAge C++ searches for a file in a long series of directories in the first part of the list, only to eventually find it in one of the last few directories, a considerable amount of time is wasted during builds. This becomes more significant when using distributed file systems such as DFS, which can significantly lengthen directory searches on platforms such as OS/2.

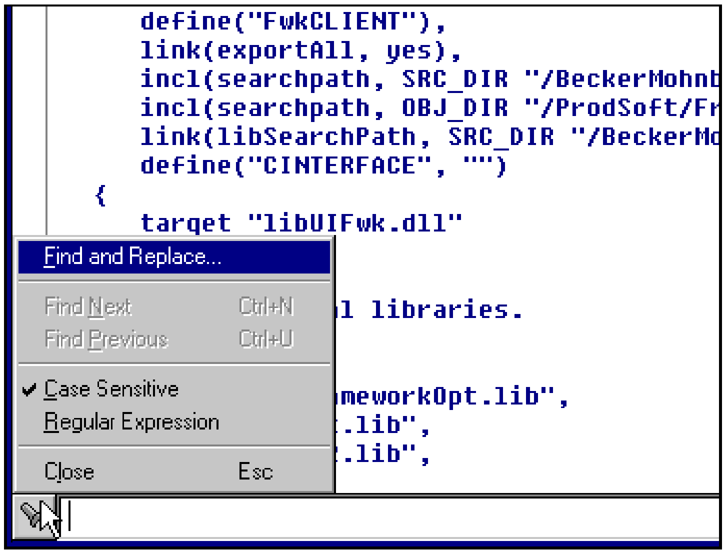
The way to avoid using a long list of incl(searchpath, *path*) directives is to have part of the directory information contained in your source directives, and have the incl(searchpath, *path*) option to point to a root directory. This way, when VisualAge C++ searches for the file, it already has several levels of directory information available to quickly locate the file. An example of this is shown in the libFramework.icc file:

```
option ProjectOptions = gen(rtti, "all")
{
   BASE_DIR = "h:/common/ActiveOrder/fwk3.5/3.5.2"
   SRC_DIR = BASE_DIR "/cmvc/src"
   GEN_DIR = BASE_DIR "/gen"
   OBJ_DIR = GEN_DIR "/Nt/obj_dbg"
   option link(linkwithmultithreadlib), link(linkwithsharedlib),
      lang(nokeyword, "bool"), lang(nokeyword, "true"),
      lang(nokeyword, "false"), define("IC_LANG_BOOL", 0),
      define("_X86_"), define("IVB_IMPORT", "_Import"),
      link(subsystem, "windows", 4, 0), incl(searchpath, "."),
      incl(searchpath, OBJ_DIR "/ProdSoft/Framework/Schema"),
      define("FwkCLIENT"), define("CICS_W32"),
      link(exportAll, yes),
   {
      target "libFramework.dll"
      {
         //
         // Common headers.
         //
         option macros('global', 'yes')
         {
            source
               "ProdSoft/Framework/Common/FwkPlatform.hpp",
               "ProdSoft/Framework/Common/FwkBigInteger.hpp",
               "ProdSoft/Framework/Common/FwkLog.hpp",
               "ProdSoft/Framework/Common/FwkMessages.hpp",
               "ProdSoft/Framework/Common/FwkTimerBase.hpp"
         }
         source type('cpp')
            "ProdSoft/Framework/Common/FwkAutoPtr.hpp",
            "ProdSoft/Framework/Common/FwkBigDecimal.hpp",
            "ProdSoft/Framework/Common/FwkCurrency.hpp"
      }
   }
}
```

12.2.3 Be aware of platform differences

The IBM OS/2 and Windows NT platforms ordinarily use the backward slash for a separator between a parent and its child directory, whereas AIX uses the forward slash character. If you are designing configuration files that are to be used on AIX and either/both OS/2 and NT platforms, then use the forward slash character. The latter two operating systems support either character, whereas AIX only supports the forward slash character.

When using the IDE on OS/2 and NT to add source files to your configuration file, you will find that the IDE may use the backward slash separator. Once you have added all the source files to the configuration file, perform a find and replace to use the forward slash separator. To perform a find and replace, open a **Live Find** in your Source view (Ctrl+F), then select the **Find and Replace** option from the menu, as shown in Figure 6.



```
        define("FwkCLIENT"),
        link(exportAll, yes),
        incl(searchpath, SRC_DIR "/BeckerMohnb
        incl(searchpath, OBJ_DIR "/ProdSoft/Fr
        link(libSearchPath, SRC_DIR "/BeckerMo
        define("CINTERFACE", "")
     {
        target "libUIFwk.dll"
                              l libraries.
                              meworkOpt.lib",
                              .lib",
                              .lib",
```

Figure 6. Find and replace a string

12.3 Splitting a project into components

The next step for the architect is to determine how each of the architectural areas is to correspond to components that can be built by the developers. In doing this, the architect will need to consider the resourcing requirements for each perimeter. This is important, as ideally each developer will want to be able to develop their source code without interfering with other developers on a frequent basis.

To do this, it would be ideal to have twice as many components as developers, so that at any point in time, a project manager has one or two components that can be assigned to each developer. And of course there are other combinations, such as one for developer A, three for developer B, and so on. The combinations will no doubt vary as the project proceeds.

The average number of components per developer may vary depending on the degree of interaction between the components. For example, there is little reason to separate one component into two components if the resulting two components would still have a high degree of interaction. In such a case, it is likely to mean that one developer will need to be developing both components.

On the other hand, a component may be split into two or more components in situations where the codestore is becoming unmanageable (for example, too much memory is required and/or too much time is required to build it). See 13.1.1, "Reducing the size of a codestore" on page 67 for a discussion on splitting one library into two libraries.

12.4 Reusing elements of configuration files

We suggest that you identify elements of your configuration files that are reusable, then place those reusable elements into a separate configuration file that can be included in your component configuration files. This can improve maintainability and make it easier to construct new component configuration files. It will be easier because the commonality will be replaced with one or more simple include directives.

12.4.1 Design points for identifying reuse

The easiest and often the best way to identify reuse is to produce several configuration files and examine them to see what is common between them. The broad areas of commonality are:

1. Compiler and linker options.
2. Include directory paths.
3. Promoted source, header, inline and template definition files. These can be categorized to subdivide a possibly long list. For example, if your components have many subsystems, categorize the files into subsystems.

12.4.2 Promoted source directives for reusable components

Imagine for the example directory structure introduced in 12.2, "Your project's directory structure" on page 54 . In this example, both applications A and B are uses of the Infrastructure.

```
{root}
    | Infrastructure
        | Middleware
        | Comms
    | Application A
        | UserInterface
        | BusinessModel
        | DatabaseAccess
    | Application B
        | UserInterface
        | BusinessModel
        | DatabaseAccess
...etcetera
```

In this scenario there will be a number of header, inline and template definition files that will be used by the Infrastructure and the two applications. The exercise you need to undertake is to determine which files are used by all of those components.

Once you have the list of files, you need to create a new configuration file that will be included within the configuration file for each of the components. The contents of the new configuration file will be a list of source directives corresponding to that list of files. Then you will edit the Infrastructure and application configuration files and add an include directive to include the new configuration file.

The following is an excerpt taken from the new configuration file *libFrameworkGroups.icc* that contains the source directives from our project, which we identified as reusable:

```
//
// Common reusable files for the two Applications.
//
option macros('global', 'yes')
{
   source type(’cpp’)
     "ProdSoft/Framework/Common/FwkBigInteger.c",
     "ProdSoft/Framework/Common/FwkBigInteger.hpp",
     "ProdSoft/Framework/Common/FwkBigInteger.inl",
     "ProdSoft/Framework/Common/FwkMessages.hpp"
}
   source type('cpp')
     "ProdSoft/Framework/Common/FwkAutoPtr.c",
     "ProdSoft/Framework/Common/FwkAutoPtr.hpp",
     "ProdSoft/Framework/Common/FwkAutoPtr.inl",
     "ProdSoft/Framework/Common/FwkBigDecimal.hpp"
```

The following excerpt is taken from one of the application configuration files that shows the new configuration file being reused:

```
target "ApplicationA.exe"
{
   include "libFrameworkGroups.icc"
   //other sources specific to this component
   //would be listed here
}
```

After performing these changes and performing a build of the two applications, you will obtain the following benefits:

- Information from both configurations is available in the IDE. You will notice in the Class page of the Project tab that you can conveniently browse the classes that you can reuse from the Infrastructure component.
- Your application configuration files will be optimized as they have all of the files that you are allowed to reuse in the Infrastructure already promoted.
- Maintenance of the three component configuration files will be made easier, as whenever a new file is added to the Infrastructure that is to be reused, then adding it to the new configuration file automatically updates the two applications.

12.4.3 Placement of reusable configuration files

Some of the reusable configuration files discussed in 12.4, “Reusing elements of configuration files” on page 57 would be best placed in the root directory of the directory tree discussed in 12.2, “Your project’s directory structure” on page 54, as it is expected those files will be reused by many of the components in the underlying directories. It is likely you will have reusable configuration files concerning the Infrastructure that are to be used by Application A and B. Place those files in the Infrastructure directory.

12.5 Organizing libFrameworkGroups.icc by grouping sources

Naming a group of source files and using that name in many configuration files instead of repeating the list of source file names becomes important in project development. In 9.3, “Changing libUIFwk.icc to build a DLL” on page 40 we discussed the build of libUIFwk.dll. This has a dependency on libFramework.dll. Library libUIFwk.dll uses header files containing classes and methods from libFramework.dll.

Typically, in this type of scenario there will be a group of header files written for the build of libFramework.dll. This group can normally be subdivided into two sub-groups as follows:

- The header files private to the build of libFramework.dll.
- The header files public to users of libFramework.dll.

We are primarily interested in grouping the public header files, although for consistency, the private header files will be done as well. To do this we need to examine libFramework.icc and determine the files that belong to each sub-group. The sub-groups will have two sub-groups themselves, as we need a split between macro and non-macro source files.

To summarize, the groups we will have are as follows:

- Private header files of libFramework.dll.
    - Macro private header files.
    - Non-macro private header files.
- Public header files of libFramework.dll.
    - Macro public header files.
    - Non-macro public header files.

12.5.1 Using groups in configuration libFramework.icc

To illustrate the use of private groups, we will change configuration libFramework.icc to use groups. To facilitate reusability, the groups will be defined in the included file libFrameworkGroups.icc.

After using the group directive, you will notice that libFramework.icc does not contain any source file names. It now purely uses the groups defined in libFrameworkGroups.icc. Hence libFrameworkGroups.icc now has all of the source file names.

The content of libFramework.icc is as follows:

```
include "libFrameworkGroups.icc"

option ProjectOptions = gen(rtti, "all")
{
   BASE_DIR = "h:/common/ActiveOrder/fwk3.5/3.5.2"
   SRC_DIR = BASE_DIR "/cmvc/src"
   GEN_DIR = BASE_DIR "/gen"
```

```
    OBJ_DIR = GEN_DIR "/Nt/obj_dbg"
    option link(linkwithmultithreadlib), link(linkwithsharedlib),
       lang(nokeyword, "bool"), lang(nokeyword, "true"),
       lang(nokeyword, "false"), define("IC_LANG_BOOL", 0),
       define("_X86_"), define("IVB_IMPORT", "_Import"),
       link(subsystem, "windows", 4, 0), incl(searchpath, "."),
       incl(searchpath, OBJ_DIR "/ProdSoft/Framework/Schema"),
       define("FwkCLIENT"), define("CICS_W32"),
       link(exportAll, yes)
    {
       target "libFrameworkOpt.dll"
       {
          source
             "libFwkCore.lib"

          SCHEMA_BIN    = SRC_DIR "\\schema.exe"
          SCHEMAGEN_DIR = OBJ_DIR "\\ProdSoft\\Framework\\Schema"

          run before
             sources("ProdSoft/Framework/Model/FwkMdlMapAttribute.s")
             targets("FwkMdlMapAttribute.cpp","FwkMdlMapAttribute.hpp",
                     "FwkMdlMapAttributeAttrib.cpp",
                     "FwkMdlMapAttributeAttrib.hpp")
             SCHEMA_BIN " -o" SCHEMAGEN_DIR " -s "
                SRC_DIR "\\ProdSoft\\Framework\\Model\\FwkMdlMapAttribute.s"
          //
          // Source directives.
          //
          option macros('global', 'yes')
          {
             source type(cpp)
                libFwkPublicM, libFwkPrivateM
          }
          source type(cpp)
             libFwkPublic, libFwkPrivate
       }
    }
}
```

You can see at the top of libFramework.icc that libFrameworkGroups.icc is included. The only two source directives used are for the macro and non-macro source files that have the Framework public and private source files.

12.5.2 Defining groups in configuration libFrameworkGroups.icc

The Framework component consists of a number of subsystems. For brevity, only two of the subsystems will be shown here. The naming convention for the groups is:

```
lib{Component}{Visibility}{Subsystem}[Macros]
```

To illustrate the use of the naming convention, we have:

- **Component**, which is *Fwk* (short for Framework).
- **Visibility** has two possible values: *Public* and *Private*. *Public* refers to source files to be exposed to users (other components) of Framework. *Private* refers to source files to be internalized within Framework.

- **Subsystem** in our example has four possible values: *System*, *OCL*, *Comms* and *Config*. The first two are system header and OpenClass files from VisualAge C++'s include directory. The last two are subsystems of Framework.
- **Macros** is optional and can have value *M*. The macro source files will be listed in a group that has suffix *M,* while non-macro source files will be listed in another group that does not have this suffix.

In our example we will have 16 groups defined for our subsystems, eight for public files and eight for private files. Four additional groups will be defined that collapse our subsystems into our component along the boundaries of public and private groups that have macro and non-macro files.

To better illustrate this, here is the content of libFrameworkGroups.icc that reflects the above groupings:

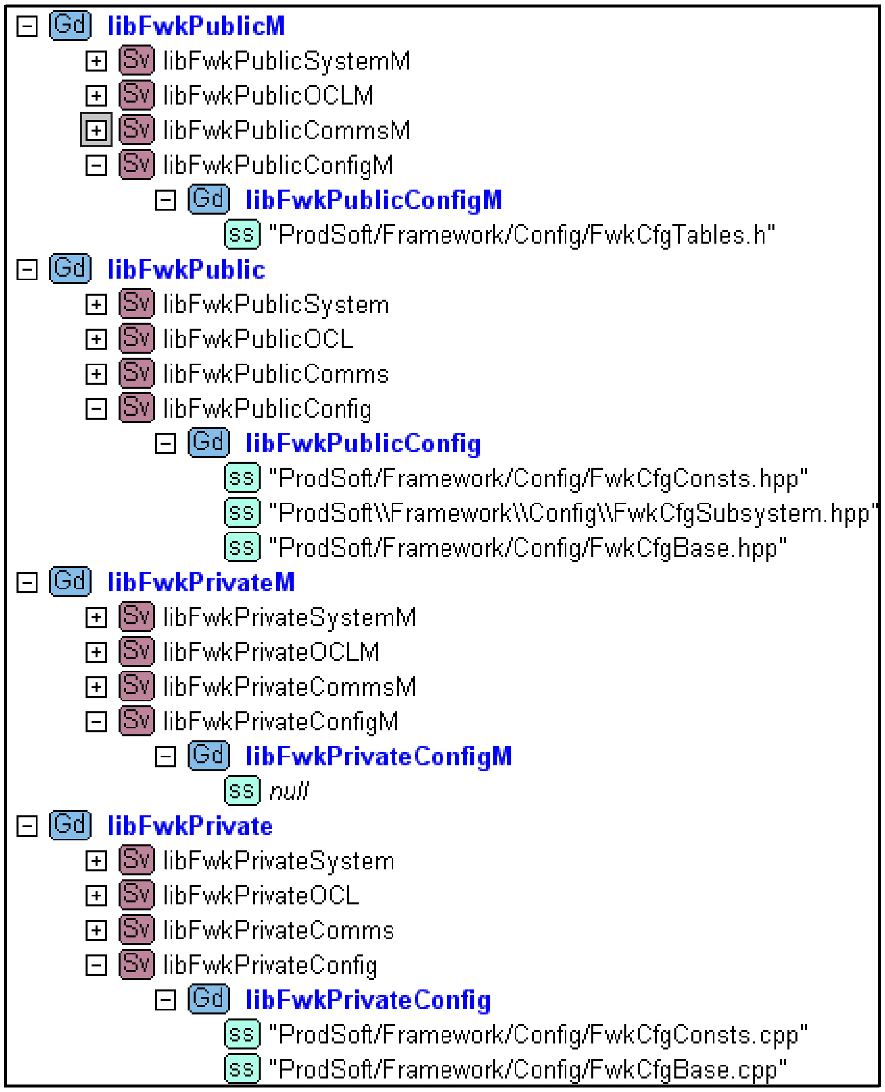
```
//
// System headers.
//
group libFwkPublicSystemM =
   "stdarg.h"
group libFwkPublicSystem =
   'fstream.h'
group libFwkPrivateSystemM =
   "io.h", "locale.h", "nl_types.h", 'iostream.h', 'sys/stat.h'
group libFwkPrivateSystem =
   "strstrea.h", "typeinfo", "typeinfo.h"
//
// OpenClass headers.
//
group libFwkPublicOCLM =
   "iseq.h", "ikb.h", 'iset.h', "iseqtab.h", "istk.h", 'ikss.h'
group libFwkPublicOCL =
   "iseq.inl", "ikb.inl", "iobservr.hpp", "iobservr.inl",
   'ihandler.hpp', 'ihandler.inl', 'iset.inl', "iseqtab.inl"
   "istk.inl", 'ikss.inl', "iobjwin.hpp", "irkeyset.h", "irkeyset.inl"
group libFwkPrivateOCLM =
   "itrace.hpp"
group libFwkPrivateOCL =
   'ievent.hpp", "ievent.inl", "ievtdata.hpp", "ievtdata.inl"
//
// Comms files.
//
group libFwkPublicCommsM = null
group libFwkPublicComms =
   "ProdSoft\\Framework\\Comms\\FwkComSubsystem.hpp"
group libFwkPrivateCommsM = null
group libFwkPrivateComms =
   "ProdSoft\\Framework\\Comms\\FwkComImplCICS.hpp",
   "ProdSoft\\Framework\\Comms\\FwkComImplCICS.cpp"
//
// Config files.
//
group libFwkPublicConfigM =
   "ProdSoft/Framework/Config/FwkCfgTables.h"
group libFwkPublicConfig =
   "ProdSoft/Framework/Config/FwkCfgConsts.hpp",
   "ProdSoft\\Framework\\Config\\FwkCfgSubsystem.hpp",
   "ProdSoft/Framework/Config/FwkCfgBase.hpp"
group libFwkPrivateConfigM = null
group libFwkPrivateConfig =
   "ProdSoft/Framework/Config/FwkCfgConsts.cpp",
   "ProdSoft/Framework/Config/FwkCfgBase.cpp"

//
// Groups.
//
group libFwkPublicM =
   libFwkPublicSystemM, libFwkPublicOCLM,
   libFwkPublicCommsM, libFwkPublicConfigM
group libFwkPublic =
   libFwkPublicSystem, libFwkPublicOCL,
   libFwkPublicComms, libFwkPublicConfig
group libFwkPrivateM =
   libFwkPrivateSystemM, libFwkPrivateOCLM,
   libFwkPrivateCommsM, libFwkPrivateConfigM
group libFwkPrivate =
   libFwkPrivateSystem, libFwkPrivateOCL,
   libFwkPrivateComms, libFwkPrivateConfig
```

Notice the groups *libFwkPublicCommsM* and *libFwkPrivateCommsM* are assigned *null*. This is because there are no macro source files for the *Comms* subsystem. Even so, they are intentionally defined as a provision for the future should there be *Comms* macro source files we need to add later.

The grouping convention might need to be violated

The macro source files are processed in the order they appear in your configuration file. There can be slight deviations from the above subsystem grouping convention depending on the dependencies between macro source files. For example, component A appears before B, and when a macro source file from B is needed before a macro source file from A, then you might not be able to have a pure split. In this situation you either need to move component B before A, or violate the grouping convention and place that macro source file from component B into component A. Attempt the former solution if the dependency is simple.

The last four groupings collapse the subsystem groups into the component level. This layering of groups into subsystem and component levels is to aid readability and make it easier when adding new source files. In the IDE you can use the Options or Targets views in the Configuration section to easily obtain a list of all the source files belonging to the Config components, as shown in Figure 7.

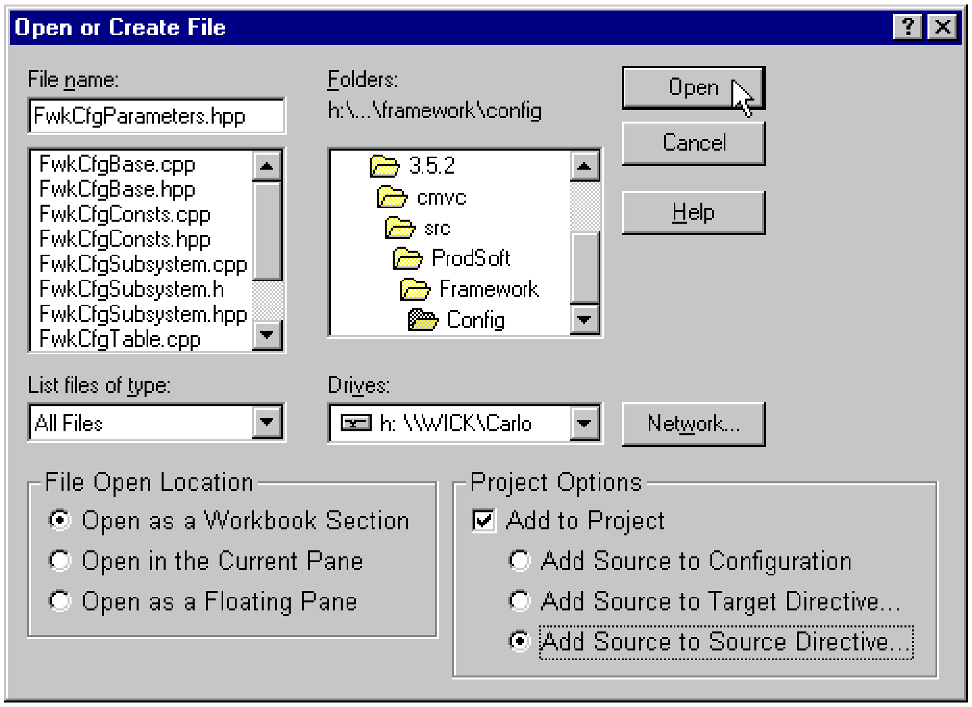


```
[-] Gd libFwkPublicM
        [+] Sv libFwkPublicSystemM
        [+] Sv libFwkPublicOCLM
        [+] Sv libFwkPublicCommsM
        [-] Sv libFwkPublicConfigM
                [-] Gd libFwkPublicConfigM
                        ss "ProdSoft/Framework/Config/FwkCfgTables.h"
[-] Gd libFwkPublic
        [+] Sv libFwkPublicSystem
        [+] Sv libFwkPublicOCL
        [+] Sv libFwkPublicComms
        [-] Sv libFwkPublicConfig
                [-] Gd libFwkPublicConfig
                        ss "ProdSoft/Framework/Config/FwkCfgConsts.hpp"
                        ss "ProdSoft\\Framework\\Config\\FwkCfgSubsystem.hpp"
                        ss "ProdSoft/Framework/Config/FwkCfgBase.hpp"
[-] Gd libFwkPrivateM
        [+] Sv libFwkPrivateSystemM
        [+] Sv libFwkPrivateOCLM
        [+] Sv libFwkPrivateCommsM
        [-] Sv libFwkPrivateConfigM
                [-] Gd libFwkPrivateConfigM
                        ss null
[-] Gd libFwkPrivate
        [+] Sv libFwkPrivateSystem
        [+] Sv libFwkPrivateOCL
        [+] Sv libFwkPrivateComms
        [-] Sv libFwkPrivateConfig
                [-] Gd libFwkPrivateConfig
                        ss "ProdSoft/Framework/Config/FwkCfgConsts.cpp"
                        ss "ProdSoft/Framework/Config/FwkCfgBase.cpp"
```

Figure 7. Using the IDE to view a list of all files belonging to a subsystem

12.5.3 Adding new files to groups

The other reason mentioned earlier was to make it easier to add new source files. Imagine that we need to create a new source file for the *Config* subsystem and it is a non-macro private source file. Now that we have our groups in place, this can easily be accompished. To do so, select the *Project Workbook* menu and then its *Open or Create File...* menu item. The *Open or Create File* dialog box will appear as shown in Figure 8.



Open or Create File

File name: FwkCfgParameters.hpp

Folders: h:\...\framework\config

- FwkCfgBase.cpp
- FwkCfgBase.hpp
- FwkCfgConsts.cpp
- FwkCfgConsts.hpp
- FwkCfgSubsystem.cpp
- FwkCfgSubsystem.h
- FwkCfgSubsystem.hpp
- FwkCfgTable.cpp

- 3.5.2
- cmvc
- src
- ProdSoft
- Framework
- Config

List files of type: All Files

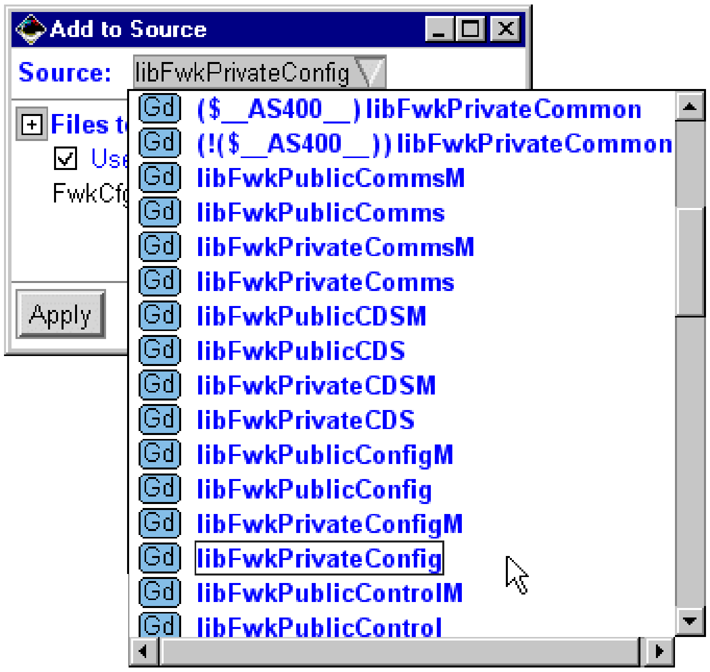
Drives: h: \\WICK\Carlo

File Open Location
- [✓ Open as a Workbook Section]
- Open in the Current Pane
- Open as a Floating Pane

Project Options
- [✓ Add to Project]
- Add Source to Configuration
- Add Source to Target Directive...
- [✓ Add Source to Source Directive...]

Figure 8. Using the Open or Create File dialog box to create a new file

In the dialog box we enter filename *FwkCfgParameters.hpp*, select the *Add to Project* checkbox, select the *Add Source to Source Directive...* radio button, and finally select the *Open* push button. The *Add to Source* dialog box will then appear as shown in Figure 9.



Add to Source

Source: libFwkPrivateConfig

- ($__AS400__) libFwkPrivateCommon
- (!($__AS400__)) libFwkPrivateCommon
- libFwkPublicCommsM
- libFwkPublicComms
- libFwkPrivateCommsM
- libFwkPrivateComms
- libFwkPublicCDSM
- libFwkPublicCDS
- libFwkPrivateCDSM
- libFwkPrivateCDS
- libFwkPublicConfigM
- libFwkPublicConfig
- libFwkPrivateConfigM
- libFwkPrivateConfig
- libFwkPublicControlM
- libFwkPublicControl

Figure 9. Adding a new source file to a source directive

We select the *libFwkPrivateConfig* group and select the *Apply* button. The result is the new non-macro, private *Config* source file *FwkCfgParameters.hpp* is added to the *libFwkPrivateConfig* group.

12.6 Organizing libFramework.icc by grouping options

You will no doubt discover that many of your components use some of the same options for compiling and linking. This means the common options can be removed from your many components and placed in a reusable configuration file, in the same way that you can group your source files.

By examining the configuration files for the spp, libFramework and libUIFwk components we can see the following common options being used:

```
incl(searchpath, "."), lang(nokeyword, "bool"), lang(nokeyword, "true"),
lang(nokeyword, "false"), define("IC_LANG_BOOL", 0), define("IVB_IMPORT",
"_Import"), link(linkwithmultithreadlib), link(linkwithsharedlib)
```

To create an option group, you use the following syntax:

option *group_name*=*list_of_options*

We can create an option group for these options named *CommonOptions* simply as follows:

```
option CommonOptions = incl(searchpath, "."), lang(nokeyword, "bool"),
lang(nokeyword, "true"), lang(nokeyword, "false"), define("IC_LANG_BOOL",
0), define("IVB_IMPORT", "_Import"), link(linkwithmultithreadlib),
link(linkwithsharedlib)
```

Then we need to place this option group into a reusable configuration file that we will call *Common.icc*. To use the option group in our spp, libFramework, and libUIFwk configuration files, we simply insert the following at the top of each of those configuration files:

```
include "Common.icc"
```

Lastly, in each of those three configuration files we replace the reusable list of options with the word *CommonOptions*. At any time in the future, if a new option needs to be applied to those three configuration files, then we simply edit the option group defined in the *Common.icc* file.

12.7 Organizing one or several targets into a configuration file

When designing your configuration files for each of your components, you will need to determine whether it is logical for some configuration files to contain more than one component (target). This is a viable decision when, for example, two components are closely coupled and only one of the components is publicly accessible by any other component in the project.

For example, combining the schema.exe and spp.exe targets into one configuration file seems like a good idea, considering that all other components only utilize schema, and schema is the only component that has a dependence (being run-time) on spp. Unfortunately, because they each define a few macros with different values, this is not technically possible. The solution was to have the two targets in separate configuration files. For more information on this, see Chapter 9, “Building an archive (static library)” on page 39.

Chapter 13. Techniques for better build efficiency

This chapter focuses on how VisualAge C++ can be used to perform more efficient builds. It shows what causes a codestore to grow excessively, and how this can be circumvented. It discusses developing with multiple projects (codestores) concurrently.

13.1 Managing system header files

As you increase the number of system header files that are included in a build, the codestore grows in size. While this offers the advantage that more information is available to you in the IDE, such as in the Classes and Declarations views, there are also the following disadvantages:

- More disk space is consumed.
- More memory is consumed.
- The time required to build increases.

13.1.1 Reducing the size of a codestore

The techniques to reduce the size of a codestore can involve one or more of the following:

1. Promote included header files (secondary source files) as primary source files to ensure that the codestore only has one image of their contents. This is perhaps the one improvement that will have the greatest impact on the efficiency of your configuration.
2. Isolate your dependence on expensive header files, such as windows.h on Windows NT and os2.h on OS/2, to use in one or a few source/header files.
3. Split libraries and executables into more libraries and have a separate codestore each.
4. Check your configuration file for any header files that contain declarations for things not used elsewhere in this configuration, and remove them.
5. Remove all #includes of header files from your source and header files.

13.1.2 Promoting #include files

Performing this task will, without doubt, give the largest optimization to your codestore. The technique to determine which files are included (secondary source files) is to use the Source Files pane of the Source Files page of the Project tab. Initially, in this pane you will see only your source files, and not those of STL/OpenClass nor system header files. You can add the last two categories of files by clicking on the filter button as shown in Figure 10, and selecting Show All.

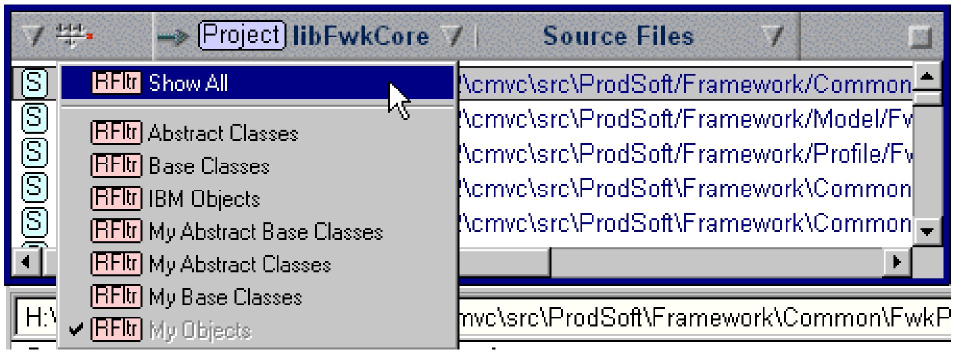

Project libFwkCore
Source Files
Show All
Abstract Classes
Base Classes
IBM Objects
My Abstract Base Classes
My Abstract Classes
My Base Classes
My Objects
\cmvc\src\ProdSoft/Framework/Common
\cmvc\src\ProdSoft/Framework/Model/Fv
\cmvc\src\ProdSoft/Framework/Profile/Fv
\cmvc\src\ProdSoft\Framework\Common
\cmvc\src\ProdSoft\Framework\Common
H:\
mvc\src\ProdSoft\Framework\Common\FwkP

Figure 10. Showing all source files in the Source Files view

Scroll down the list until you start seeing files with the 'I' glyph. To begin with, ensure all your application source files are promoted, then start on the OpenClass and/or system header files.

Before you promote a secondary source file, you should check whether it defines any macros that are used by other primary source files. If so, then the option macros('global','yes') needs to be applied to that source file. Otherwise, the macros defined in that file will only be known within the scope of that file.

To determine if a source file contains any macros can be performed reasonably easily, although tediously: select the source file and in the editor perform a find for *#define*. If the source file defines macros other than the guard define which prevents multiple reinclusion of a header file, then it should be promoted as a macro source file.

Be careful with the order in which you list the macro source files in your configuration file. They will be processed in the order in which they are listed. The order becomes important when a macro source file uses a macro from another macro source file. The latter macro source file needs to be listed before the former; otherwise, an error will be produced during build.

We do not recommend that you promote **all** secondary source files. There are likely to be too many files in the average project for you to easily check every one. There will also come a point where promoting more files will make little difference to codestore size. Instead, promote those which you believe are included more than once, as it is only those files which will have a significant impact on the size of your codestore.

13.1.3 Isolating dependence on expensive header files

On the Windows NT and OS/2 platforms, the windows.h and os2.h files are expensive header files, as they include many other header files and accumulate a great number of declarations. For example, on the Windows NT platform, the following configuration file generates a codestore of size 4,702,575 bytes.

```
option macros(global, yes)
{
   source type(cpp) "windows.h"
}
```

Compare this to a simpler header file such as stdio.h that generates a codestore of size 242,178 bytes.

Now we will find where windows.h is being used and isolate our dependence on it. The method to do this involves:

- Searching for the text *windows.h* by using the Search page of your Project tab and determining which of the source and header files you found use things declared in windows.h.
- Modifying your found header files to use forward declarations if the full declaration is not required. This will eliminate the need to include windows.h.
- Removing windows.h from your configuration file and from being included in your source and header files.
- Consolidating all uses of declarations in windows.h to one or a few source and header files. This will become the basis for an interface to services provided by windows.h.

Once you have completed the above, performing a build will likely fail as your files still have a dependence on windows.h. All we have done is to isolate the dependence to a small set of source and header (isolated) files. See 13.1.4, “Moving source and header files into a separate library” on page 69 for instructions on building the isolated files into a separate library.

13.1.4 Moving source and header files into a separate library

For *libFramework.dll* we found that a few source and header files used windows.h. We removed those files from libFramework.icc and added them to our new configuration file called libFwkCore.icc.

The content of libFwkCore.icc is as follows:

```
option ProjectOptions = gen(rtti, yes)
{
   BASE_DIR = "h:/common/ActiveOrder/fwk3.5/3.5.2"
   SRC_DIR = BASE_DIR "/cmvc/src"
   GEN_DIR = BASE_DIR "/gen"
   OBJ_DIR = GEN_DIR "/Nt/obj_dbg"
   option link(linkwithmultithreadlib),
      incl(searchpath, "."), lang(nokeyword, "bool"),
      lang(nokeyword, "true"), lang(nokeyword, "false"),
      define("IC_LANG_BOOL", 0), define("_X86_"),
      define("IVB_IMPORT", "_Import"),
      define("FwkCLIENT"), define("_Export", ""),
      incl(searchpath, OBJ_DIR "/ProdSoft/Framework/Schema")
   {
      target "libFwkCore.lib"
      {
         option macros('global','yes')
         {
            source "winsock2.h"
         }
         option macros('global','yes')
         {
            source
               "ProdSoft\\Framework\\Interface\\Applications\\FwkExtTCP.hpp",
               "ProdSoft\\Framework\\Interface\\Applications\\FwkExtWeb.hpp",
               "ProdSoft\\Framework\\Common\\FwkPlatform.hpp"
         }
         source type('cpp')
```

```
            "ProdSoft\\Framework\\Interface\\Applications\\FwkExtTCP.cpp",
            "ProdSoft\\Framework\\Interface\\Applications\\FwkExtWeb.cpp",
            "ProdSoft/Framework/Common/FwkPlatform.cpp",
            "ProdSoft/Framework/Model/FwkMdlKeyFactory.cpp",
            "ProdSoft/Framework/Profile/FwkUprDCE.cpp",
            "ProdSoft\\Framework\\Model\\FwkMdlKeyFactory.hpp",
            "ProdSoft\\Framework\\Profile\\FwkUprDCE.hpp"
        }
    }
}
```

For the above source files, the level of dependence on windows.h is as follows:

- FwkExtTCP.cpp and FwkExtTCP.hpp were removed from libFramework.icc, because they use windows.h but are not used by anything else.
- FwkExtWeb.hpp does not use windows.h but is used by libFramework.dll, hence it remains in libFramework.icc.
- FwkExtWeb.cpp uses FwkExtTCP.hpp and is no longer used by libFramework.dll, hence it was removed from libFramework.icc. Remember that FwkExtTCP.hpp uses windows.h, thus, for libFramework.icc to lose its dependence on windows.h, FwkExtWeb.cpp had to be moved to libFwkCore.icc to be with FwkExtTCP.hpp.
- FwkPlatform.hpp was including winsock2.h and is used by libFramework.icc. Hence the #include of winsock2.h was removed. The only thing used from winsock2.h by libFramework.icc was from an embedded #include file that declared a typedef for BYTE. This typedef was copied and pasted into FwkPlatform.hpp in place of the include of winsock2.h.
- FwkMdlKeyFactory.cpp and FwkUprDCE.cpp use windows.h, hence they were removed from libFramework.icc.
- FwkMdlKeyFactory.hpp and FwkUprDCE.hpp do not use windows.h, but they were used by libFramework.icc, hence they remained.

Code in libFwkCore cannot be debugged

The integrated debugger in VisualAge C++ only uses the codestore for debug information. This means that if you have an executable that calls functions in a DLL, you built in another codestore, you cannot use the integrated debugger to debug the functions called in your DLL. The alternatives are to build the two targets into one codestore, or to use the option link(debug, yes) for both targets and use the debugger from C and C++ Compilers Version 3.6.5 on Windows NT.

The last step to make this work was to have *libFramework.dll* link against our new static library. This was easily accomplished by adding the following source directive to libFramework.icc within the target directive for libFramework.dll:

```
source "libFwkCore.lib"
```

Figure 11 shows the revised organization of the project after splitting libFramework.dll into two libraries.

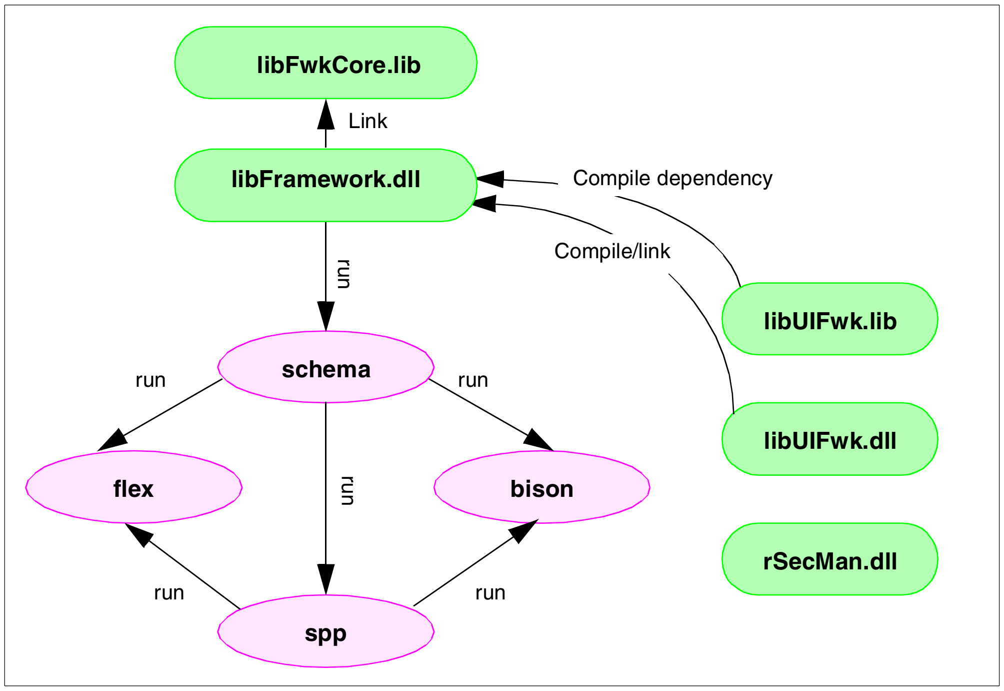


Figure 11. The new structure of the project

Be careful of moving source code with members to be exported

Some source code in libUIFwk.dll references the source code compiled into libFwkCore.lib. The library libFwkCore.lib is linked into libFramework.dll. Hence libFramework.dll now contains the compiled source code of libFwkCore.lib. Because libUIFwk.dll links against libFramework.dll to resolve external references, the source code in libFwkCore.lib needs to be exported in order for libUIFwk.dll to resolve all its external references.

If you are using AIX, this cannot be done, but it can be done on OS/2 or Windows NT. It cannot be done on AIX because the archive does not contain information on which members have been exported, whereas OS/2 and Windows NT do contain this information in the import library.

To flag members in libFwkCore.lib to be exported from libFramework.dll, it is easiest to modify the source code that is compiled into libFwkCore.lib. Use the *_Export* keyword as part of your class or member declarations. Use the keyword at the class scope to export all members of the class. Otherwise, use it at the member scope if you need to export a subset of all members of a class.

13.1.5 Removing files that are no longer referenced

It is possible over time that your configuration files may include source and/or header files that are no longer used. This can occur due to the changing nature of your source code as time progresses. Hence, you may want to periodically inspect your configuration files to see if you can determine, at a glance, any files that are no longer referenced, and remove them from your configuration file.

When applying the technique described in 13.1.4, “Moving source and header files into a separate library” on page 69 you will need to remove files that are no longer referenced. It is important to perform this immediately, otherwise it is likely you will forget to do this in the future, and you will not obtain all the advantages you sought.

13.1.6 Removing #include from all your source files

Carrying out this technique helps you to determine which secondary source files you are directly including in application files that you have not promoted. These source files will likely be the ones that, if promoted, will give your build a large boost in build performance.

Building your code on non-VisualAge C++ platforms

As orderless programming is not part of the C++ standard, be careful if you choose to remove #include directives from your source files. Only the VisualAge C++ Version 4.0 compiler can compile your source code without #include directives.

13.1.7 Results of applying these techniques in our project

In Chapter 8, “Building a shared (dynamic link) library” on page 35, we explored building a Dynamic Link Library (DLL) named libFramework.dll. For this example we applied all of the points listed at the beginning of this chapter.

13.1.7.1 Results of applying the isolation technique

The results of isolating and moving out the source files with dependence on windows.h are shown in Table 4. The most significant reductions were for the codestore file and memory use. The advantage of having a smaller codestore file is the saving in disk space and speed in saving the codestore file. If a number of developers are working on its contents and not that of libFwkCore, then the latter’s codestore is not needed by them. Reducing the amount of memory being utilized means that memory constrained systems will perform better, and the reduction observed for clean build times would be greater if mild to heavy memory paging occurs.

Table 4. Results of applying first two codestore optimization techniques

Affected area	Original libFramework	Optimized libFramework	Reduction
codestore file size	27,736,507 bytes	23,305,848 bytes	16%
IDE memory utilization	85,044 kilobytes	77,864 kilobytes	8.5%
clean build time	2m 40s	2m 31s	5.5%

13.1.7.2 After applying all techniques

After applying the last three techniques above, further reductions were obtained. The reductions were not as great after applying the first two techniques. This is largely because the configuration file to begin with already had included files promoted, as described in 13.1.2, “Promoting #include files” on page 67.

The new results are shown in Table 5. These reductions are being compared to the values that existed before any of the techniques were applied above.

Table 5. Results of applying the five codestore optimization techniques

Affected area	**Original libFramework**	**Optimized libFramework**	**Reduction**
codestore file size	27,736,507 bytes	23,296,918 bytes	16%
IDE memory utilization	85,044 kilobytes	73,652 kilobytes	13.5%
clean build time	2m 40s	2m 15s	15.5%

13.2 Developing multiple projects concurrently

You will no doubt find that if you have multiple projects that you are developing concurrently, then opening one project, closing it, and then opening another repeatedly can be a time-consuming exercise. If your computer has a sufficient amount of memory and paging space, then you may find it more productive to open a separate session of VisualAge C++ for each of your projects. This way you can rapidly switch between projects, and you also have the advantage of leaving the views of each project’s tabs, pages, and panes unaffected after the switch.

13.3 Automating a project’s build process

Trivial projects that involve one component will not require a build process. When a project grows and there are a multitude of components, then a build process is required. The build process will coordinate the builds of each component when the entire project needs to be built.

In automating the build process, the IDE cannot be used, as it requires user input. Instead, a batch-oriented process is needed, and this can be achieved by using a script file or even a makefile. To perform builds in batch mode, the *vacbld* command must be used. This command takes as input a configuration file and runs the incremental compiler to produce a codestore. Provided there are no build errors, it will also produce the targets defined in the configuration file.

The *vacbld* command can be invoked simply as follows:

```
vacbld libFramework.icc
```

13.3.1 Building our project components collectively

We will develop a configuration file that is able to build all the components of our small software project. First we will quickly recap the content of our project, before delving into building them as part of a build process. To see in detail the composition of our project, see Chapter 2, “About the sample project” on page 7. Our small software project consists of:

- Two executables named schema.exe and spp.exe. One does not need to be built before the other. However, both need to be built before proceeding with building libFramework.dll.
- A static library, libFwkCore.lib.

- A dynamic link library, (DLL) libFramework.dll, that depends directly on schema.exe and libFwkCore.lib.
- A static library, libUIFwk.lib, that depends on libFramework.dll having been built because it needs some of the generated files from the latter's build.
- A resource library, rSecMan.dll, that has no dependencies on the other components.

As can be determined from the above project composition, we have a number of dependencies that need to be factored in. We will use the *run* directive to build each component using *vacbld*. We will use the *run* directive's *sources* and *targets* parameters to implement the dependencies between our components.

13.3.2 Configuration file to implement our build process

The configuration file *BuildProcess.icc* is shown below. Notice its use of the *run* directive to build our components, and how the correct source and target parameters are specified to implement the required component dependencies. The *cleanup* parameter is also used to enable a clean environment to be set up before a clean build of the entire project. Ideally these directives would be placed in each of the component configuration files and executed using the *cleanup* option of *vacbld*. For simplicity, we have not done this.

```
//
// Setup the build environment.
//
BUILD_DIR = "h:\\common\\ActiveOrder"

////////////////////////////////////////////////////////////////////////
//
// Build the components under the Application directory tree.
//
////////////////////////////////////////////////////////////////////////

//
// Build rSecMan.dll.
//
run targets ("rSecMan.dll")
    "echo Build rSecMan.dll",
    "cd " BUILD_DIR "\\C11u\\bld\\cmvc\\src",
    "vacbld rSecMan.icc"
//the run cleanup directive will only be executed when you select
//"Clean" from the IDE's Project Workbook menu.
run cleanup
    "cd " BUILD_DIR "\\C11u\\bld\\cmvc\\src",
    "del rSecMan.ics rSecMan.dll"

////////////////////////////////////////////////////////////////////////
//
// Build the components under the Framework directory tree.
//
////////////////////////////////////////////////////////////////////////

//
// Build spp.exe.
//
run targets ("spp.exe")
```

```
    "echo Build spp.exe",
    "cd " BUILD_DIR "\\fwk3.5\\3.5.2\\cmvc\\src",
    "vacbld spp.icc"
run cleanup
    "cd " BUILD_DIR "\\fwk3.5\\3.5.2\\cmvc\\src",
    "del spp.ics spp.exe"
//
// Build schema.exe.
//
run sources ("spp.exe")
    targets ("schema.exe")
    "echo Build schema.exe",
    "cd " BUILD_DIR "\\fwk3.5\\3.5.2\\cmvc\\src",
    "vacbld schema.icc"
run cleanup
    "cd " BUILD_DIR "\\fwk3.5\\3.5.2\\cmvc\\src",
    "del schema.ics, schema.exe"
//
// Build libFwkCore.lib.
//
run targets ("libFwkCore.lib")
    "echo Build libFwkCore.lib",
    "cd " BUILD_DIR "\\fwk3.5\\3.5.2\\cmvc\\src",
    "vacbld libFwkCore.icc"
run cleanup
    "cd " BUILD_DIR "\\fwk3.5\\3.5.2\\cmvc\\src",
    "del libFwkCore.ics libFwkCore.lib"
//
// Build libFramework.dll.
//
run sources ("schema.exe", "libFwkCore.lib")
    targets ("libFramework.dll")
    "echo Build libFramework.dll",
    "cd " BUILD_DIR "\\fwk3.5\\3.5.2\\cmvc\\src",
    "vacbld libFramework.icc"
run cleanup
    "cd " BUILD_DIR "\\fwk3.5\\3.5.2\\cmvc\\src",
    "del libFramework.ics libFramework.dll libFramework.lib"
//
// Build libUIFwk.dll.
//
run sources ("libFramework.dll")
    targets ("libUIFwk.dll")
    "echo Build libUIFwk.dll",
    "cd " BUILD_DIR "\\fwk3.5\\3.5.2\\cmvc\\src",
    "vacbld libUIFwk.icc"
run cleanup
    "cd " BUILD_DIR "\\fwk3.5\\3.5.2\\cmvc\\src",
    "del libUIFwk.ics libUIFwk.dll libUIFwk.lib"
```

13.3.3 Problems with executing our build process

Several problems were encountered with building our project using the configuration file BuildProcess.icc. The entire project was built within the Integrated Development Environment (IDE). The problems are as follows:

- Performing a clean build of rSecMan.dll (to cause it to be rebuilt) produces a segmentation fault in the IDE. The work-around was to build it separately and then build the entire project. See 10.5, “Problems with building the resource library” on page 45 for further details on what was causing the problem.
- If the build for one of the components did not complete due to errors, then a segmentation fault occurs in the IDE. The solution was to:
    - Open the project for the component that failed.
    - Fix the build errors.
    - Close the IDE.
    - Delete the codestore BuildProcess.ics (not the codestore for the failed component).
    - Restart the IDE and perform a build. The failed component will attempt to build, starting where it finished earlier.
- The working directory is not changed to the directory where each of the component configuration files reside. The workaround is to manually change the working directory before executing *vacbld*.

Chapter 14. Techniques for more efficient C++ programming

This chapter focuses on how VisualAge C++ can be used to perform more efficient C++ development. It shows how the Integrated Development Environment (IDE) can be used to locate certain types of inefficiencies, and how you can replace them with an efficient equivalent. It explains what causes a codestore to grow excessively and how this can be circumvented. It discusses how to better manage your configuration files to make them more readable and easier to maintain, as well as development with multiple projects (codestores) concurrently.

14.1 Find and eliminate literal strings used with IString

One of the IDE's strong features is the Find Uses view, described in Chapter 15.6, "Search for objects, not for strings" on page 86. This allows you to locate all sorts of objects, such as classes and methods.

You can take advantage of this to find uses of the IString constructor that take a const char* as parameter so that you can find a large number of IString objects constructed with a literal string. You can identify many IString objects constructed with the same literal string and replace these with one static IString object. There are several advantages to doing this:

- Executable/library size can be reduced, because an IString constructor call is eliminated.
- Run-time performance can be improved after startup, because the replacement static IString object is constructed only once during startup.
- Maintainability can be improved because the replacement static IString is declared and defined in one place instead of being spread out amongst many files.
- Run-time memory usage can be reduced because the executable/library size is smaller.

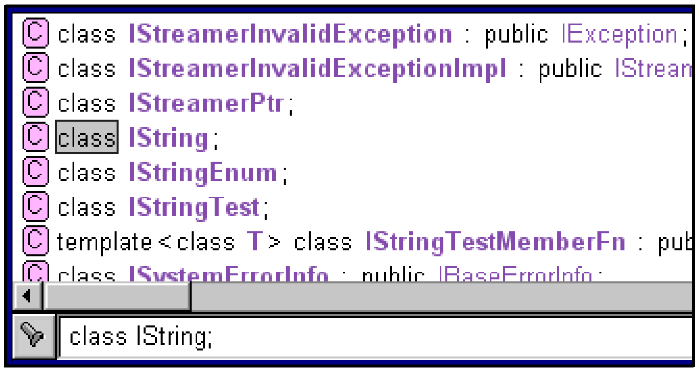
Standard Template Library (STL) equivalent of IString

If you are using *STL* and not *OpenClass,* then you will no doubt be using the equivalent class to *IString* being *std::string*. The technique described in this chapter can be applied to *std::string*.

There are several ways to search for uses of the *IString* constructor *IString(const char*)* in the IDE. One method involves using the *Classes* page of the *Project* tab. To do this, perform the following steps in order:

1. Select the **Project** tab.
2. Select the **Classes** page.
3. Select the **Filter** button on the **Classes** pane.
4. Select **IBM Objects** from the filter pop-up menu.
5. Either use the Live Find (press **Ctrl+F**) and search for `class IString;` or scroll down the list of classes until you see the IString class. Figure 12 on page 78 shows how the former search is performed.

6. In the **Members** pane, move the mouse pointer over the `IString(const char*)` constructor and press the right mouse button once. A pop-up menu will appear.
7. Select **Open as a Workbook Section** menu item to open that constructor onto its own Workbook tab.
8. Select the **Find Uses** page to see all uses of that constructor.

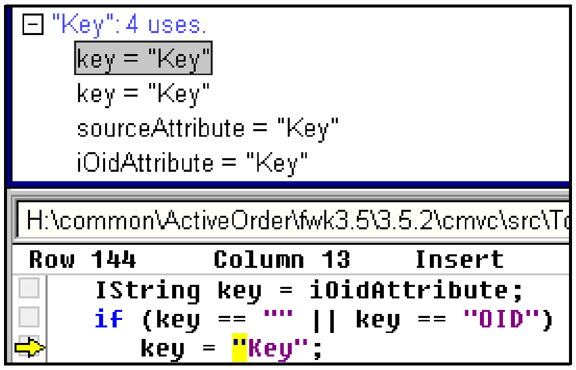


```
class IStreamerInvalidException : public IException;
class IStreamerInvalidExceptionImpl : public IStrean
class IStreamerPtr;
class IString;
class IStringEnum;
class IStringTest;
template < class T > class IStringTestMemberFn : pub
class ISystemErrorInfo : public IBaseErrorInfo;
```

class IString;

Figure 12. Searching for IString in the classes pane

As an example, the build of *schema.exe* shows that there are 4 uses involving the string *"Key"* as shown in Figure 13 on page 78. We will replace those uses with a *static const IString named strKey*. To do this, we will create a new source file named *SchemaConsts.cpp* and add it to the *schema.icc* configuration file.



```
"Key": 4 uses.
    key = "Key"
    key = "Key"
    sourceAttribute = "Key"
    iOidAttribute = "Key"

H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\T
Row 144      Column 13      Insert
    IString key = iOidAttribute;
    if (key == "" || key == "OID")
       key = "Key";
```

Figure 13. Finding uses of a string

The content of our new file *SchemaConsts.cpp* is as follows:

```
class SchemaConsts
{
   public:
      static const IString strKey;
   private:
      SchemaConsts(); // prevent construction.
};

const IString SchemaConsts::strKey("Key");
```

We then need to replace the use of *"Key"* with *SchemaConsts::strKey*. This is easily accomplished within the IDE by selecting each of the uses and replacing the string *"Key"* with *SchemaConsts::strKey* using the editor. After performing the four changes, the size of *schema.exe* reduced from 439,808 to 438,784 bytes. This is not a significant saving, but there can be hundreds of these in typical applications, and so a reduction of several percent can easily be achieved.

Performing a clean build may produce a different size target

In the above example of the optimization of *schema.exe*, the incremental build produced an executable of 439,296 bytes in size. When we performed a clean build, the executable was further reduced to 438,784 bytes. There is a minor defect that causes this discrepancy. The workaround to produce the target at the same size repeatedly is to perform clean builds. This will no doubt occur when producing your application from an automated build process.

Further similar changes and others described in 14.2, “Find and eliminate temporary copies of objects" were made, and *schema.exe* was further reduced from 439,808 to 431,104 bytes. This represents a 2% reduction. The amount of reduction obviously depends on how well your C++ programmers know the inefficiencies of C++ programming. For instance, with *libFramework.dll* we achieved a more significant reduction of 15%, and there was still more room for improvement.

14.2 Find and eliminate temporary copies of objects

The types of changes that were made to *schema.exe* are described in Table 6 on page 79. This table refers to *strName*, its type is *IString*. When *strName* is used in the context of the class *MyClass,* it is a member of *MyClass*. The table is not exhaustive, but gives you a quick reference on writing more efficient C++ code.

Table 6. Examples of writing more efficient C++ code

Before change	After change	Reason for change
strName = "";	strName = SchemaConsts::strEmpty;	Eliminate a temporary IString being constructed.
if (strName != "")	if (strName.length() != 0)	Eliminate a temporary IString being constructed.
if (strName == "")	if (strName.length() == 0)	Eliminate a temporary IString being constructed.
IString strName = "";	IString strName;	Eliminate a temporary IString being constructed and copied.

IString strNum;
strNum = strName;
	IString strNum(strName);	Eliminate calling default IString constructor.

const IString& func();
IString strName(func());
	const IString& func();
	const IString& strName(func());
		Eliminate strName being constructed as a copy of the IString returned from func(). This can only be done if strName is used as a const.

Before change	After change	Reason for change

MyClass::MyClass() {
strName = "-";
}
	MyClass::MyClass()
	: strName("-") {
	}
		Eliminate calling default IString constructor.

MyClass::MyClass() {
strName = "";
}
	MyClass::MyClass() {
	}
		Eliminate a temporary IString being constructed.

MyClass::MyClass(const IString& aName) {
strName = aName;
}
	MyClass::MyClass(const IString& aName) :
	strName(aName) {
	}
		Eliminate calling default IString constructor.

catch(IException exc)	catch(IException& exc)	Eliminate a temporary IException being constructed.

func(const IString& aName);
func("some string");
	func(const char* pszName);
	func("some string");
		Eliminate a temporary IString being constructed by all users of func(). If most users of func() pass an IString object, then do not perform the change.

// strPlace, strColor, a, b, x, y
// and z are of type IString.
strName = a + b + x;
strPlace = a + b + y;
strColor = a + b + z;
	IString strAB(a + b);
	strName = strAB + x;
	strPlace = strAB + y;
	strColor = strAB + z;
		Eliminate two concatenations of strings a and b.

14.3 Other useful references

For more detailed information on object-based searching in the IDE, see Appendix 15.6, “Search for objects, not for strings” on page 86.

Other tips on optimizing your code are provided in the VisualAge C++ Migration User’s Guide and Reference, available at:

```
www.software.ibm.com/ad/visualage_c++/downloads.html.
```

Chapter 15. Working in the IDE

The first time you work with it, the vast array of functions in VisualAge C++ may seem intimidating. There are, however, many useful and powerful features that are worth learning. In this chapter, we discuss some of these features.

15.1 See how your source code was interpreted

A fast way to see how your code looks after preprocessing is to use a Token Stream view of your source file.

The Token Stream view shows your source code with all macros or variables resolved to runtime values. This can also be a handy way to skim through the logic in your source code, as it does not display #includes and comments.

This is most useful when you have to trace the source of an error message. If you select a message in the Messages view and click mouse button 2 over it, you will have the option of opening the message as a workbook section. When you do this, you will see a page showing a Token Stream View and a source view of the code that generated the message.

You can change the view for any code object (such as a function, message, file, class, and so on) in any pane to a Token Stream view.

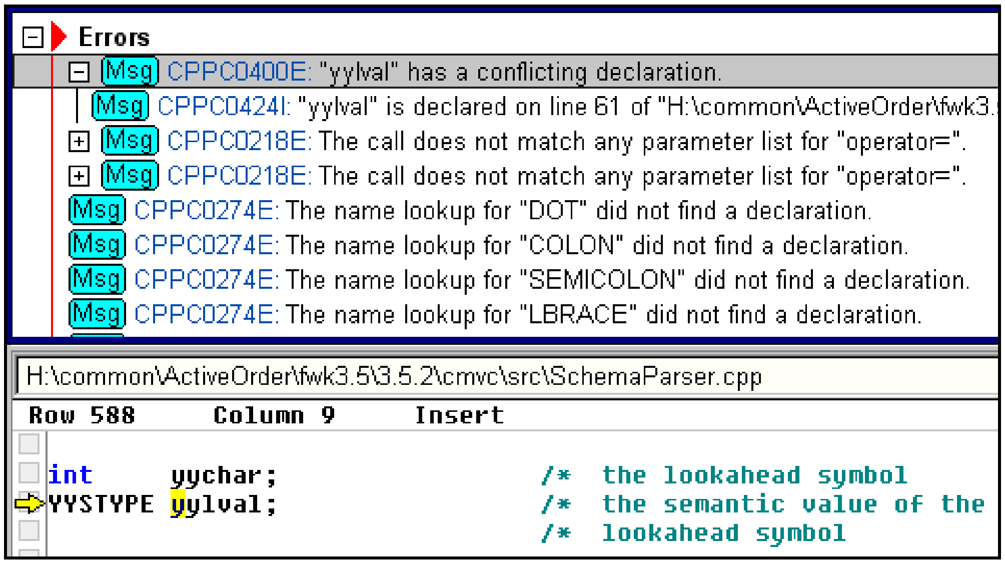
Examples of the Token Stream view appear in the next section, 15.2, “Using the IDE to investigate build errors” on page 81.

Note: You cannot open a Token Stream view for a configuration file. You can however, see an "Interpreted" view, which is similar. Read 15.7, “See how your configuration file was interpreted” on page 87.

15.2 Using the IDE to investigate build errors

Our sample project includes a component, schema.exe, which uses two source files that define a macro with different values. SchemaParser.cpp and SchemaParser.hpp both define the macro YYSTYPE. However, we eventually found that we needed to use the value defined in SchemaParser.cpp. Here is an example in which we encountered a build error, traced the source of the error, and corrected the error, using some of the views in the IDE.

At first, we built our source code with both files promoted as non-macro source files. We got the build errors shown in Figure 14.

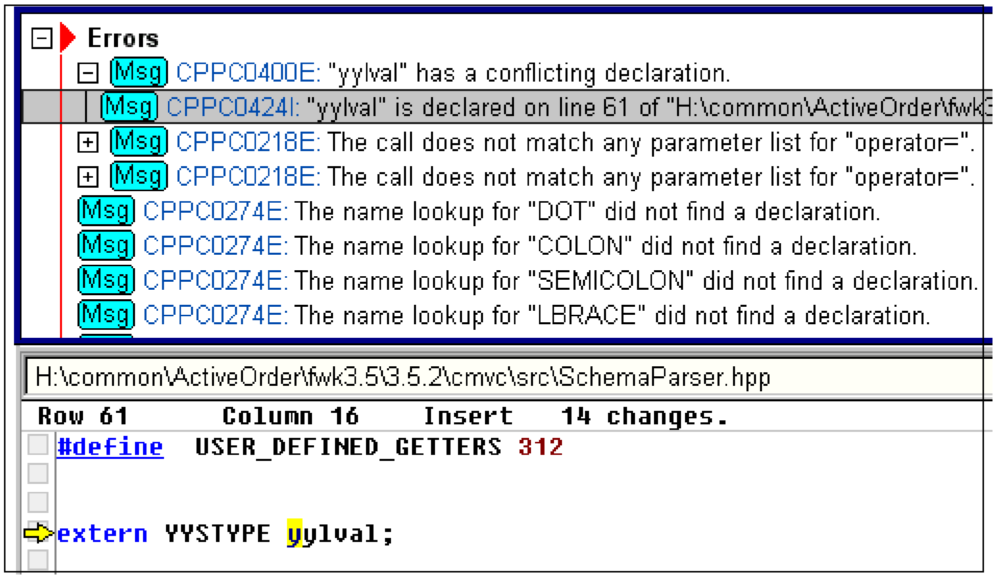


```
Errors
  Msg CPPC0400E: "yylval" has a conflicting declaration.
    Msg CPPC0424I: "yylval" is declared on line 61 of "H:\common\ActiveOrder\fwk3.
  Msg CPPC0218E: The call does not match any parameter list for "operator=".
  Msg CPPC0218E: The call does not match any parameter list for "operator=".
  Msg CPPC0274E: The name lookup for "DOT" did not find a declaration.
  Msg CPPC0274E: The name lookup for "COLON" did not find a declaration.
  Msg CPPC0274E: The name lookup for "SEMICOLON" did not find a declaration.
  Msg CPPC0274E: The name lookup for "LBRACE" did not find a declaration.
```

```
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\SchemaParser.cpp
Row 588      Column 9      Insert

int       yychar;                  /*  the lookahead symbol
YYSTYPE yylval;                    /*  the semantic value of the
                                   /*  lookahead symbol
```

Figure 14. Identifying a compiler error due to a conflicting declaration

We selected the first message. The source view below the messages showed us the conflicting declaration (Figure 15). The fact that one declaration is an *extern,* and the other is not, is unimportant. What is important is to find out what *YYSTYPE is*.



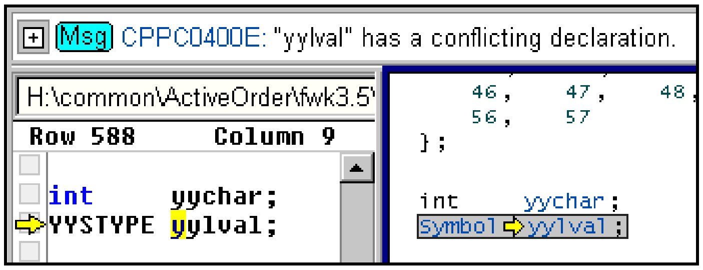
```
Errors
  Msg CPPC0400E: "yylval" has a conflicting declaration.
    Msg CPPC0424I: "yylval" is declared on line 61 of "H:\common\ActiveOrder\fwk3
  Msg CPPC0218E: The call does not match any parameter list for "operator=".
  Msg CPPC0218E: The call does not match any parameter list for "operator=".
  Msg CPPC0274E: The name lookup for "DOT" did not find a declaration.
  Msg CPPC0274E: The name lookup for "COLON" did not find a declaration.
  Msg CPPC0274E: The name lookup for "SEMICOLON" did not find a declaration.
  Msg CPPC0274E: The name lookup for "LBRACE" did not find a declaration.
```

```
H:\common\ActiveOrder\fwk3.5\3.5.2\cmvc\src\SchemaParser.hpp
Row 61       Column 16     Insert    14 changes.
#define  USER_DEFINED_GETTERS 312


extern YYSTYPE yylval;
```

Figure 15. Identifying the conflicting declaration

YYSTYPE contains all uppercase characters. This was a hint that it was a macro. However, we were not sure that it was a macro. If we used the Macros view, we would see only those macros contained in files already promoted as macro sources. Instead, we used the Token Stream view to see what value *YYSTYPE* had (if it was indeed a macro) for each of the two declarations.

To access the Token Stream pane for each declaration, we positioned the mouse pointer over each message and double-clicked. (You could also click the right mouse button and select *Open as a Workbook Section* from the pop-up menu.) For the first declaration, the value of *YYSTYPE* is *Symbol* shown in Figure 16:

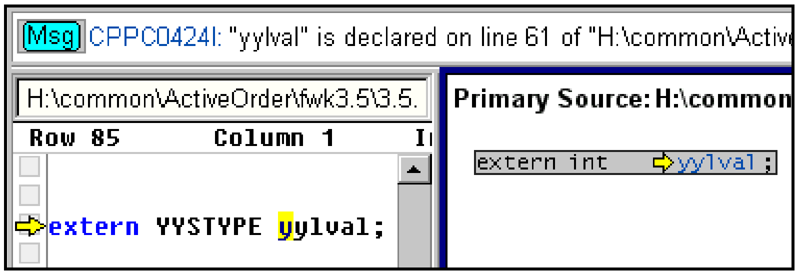


```
Msg CPPC0400E: "yylval" has a conflicting declaration.
H:\common\ActiveOrder\fwk3.5\
Row 588     Column 9
int      yychar;
YYSTYPE  yylval;
```

```
    46,    47,    48,
    56,    57
};

int    yychar;
Symbol yylval;
```

Figure 16. Value of YYSTYPE in first conflicting declaration

For the second declaration, the value of *YYSTYPE* is *int,* as shown in Figure 17.



```
Msg CPPC0424I: "yylval" is declared on line 61 of "H:\common\Activ
H:\common\ActiveOrder\fwk3.5\3.5.
Row 85      Column 1
extern YYSTYPE yylval;
```

```
Primary Source: H:\common
extern int    yylval;
```

Figure 17. Value of YYSTYPE in second conflicting declaration

Notice that the values are different for the two declarations. This is why we were getting a compiler error. By looking further into this, we found out that SchemaParser.cpp and SchemaParser.hpp both define *YYSTYPE,* and that the latter would only define it, if it was not already defined. We also knew that we wanted the definition fromSchemaParser.cpp. The solution was to promote SchemaParser.cpp as a macro source file.

15.3 Quickly see and promote your included files

A fast way to identify the source files that are currently considered secondary sources is to use a Source Files view of your project (use the Source Files page in the Project section).

Files that are considered primary source files are shown with an S graphic. Those that are #included are shown with an I graphic. To promote the included files, select one or more of those shown with an I, click mouse button 2, and select **Add source to configuration**, **Add source to source directive**, or **Add source to target directive** from the pop-up menu. If you select **Add source to configuration**, the source file will appear in a separate directive at the top of the configuration file (that is, it will not be added to the list of source files associated with any target).

By default, this view is filtered to show you only the files that are considered your files. Files located in any of the directories created by the VisualAge installation are hidden. If you remove the filter, you will see all the files that are built.

If you want to promote system files to your configuration, remove this filter by clicking on the filter symbol in the pane title bar and selecting **Show All** from the list of filters.

With the filter removed, you can see all the files that are built in your configuration, including standard library header files.

15.3.1 Notes on C source files

- Files that are included by sources that are processed as C files will still appear as 'I' objects in the Source Files view, even if you promote them in the way described above.
- The Source Files view can also serve as a fast way to locate those files that are not currently recognized as compilable by the C++ compiler. For example, if you have a primary source file called "testfile.c", but do not have the type clause type('cpp') in your source directive, this file will be displayed with a 'ExtSrc' graphic. This means that the compiler is launching an 'extension' (in this case, the C compiler) to compile this file. This is much slower on incremental builds than using the C++ compiler, as compiled C code is not stored in the codestore. Locate these files in the configuration, and add the appropriate type clause to the source directive to prevent this from happening.

15.4 Which files include other files?

Note: This feature is not available on VisualAge C++ for AIX.

While the Source Files view gives you a quick list of included files, it does not tell you *which* files include which other files. In VisualAge C++ for OS/2 and Windows, there is another, graphical view to show you this information: the Include Hierarchy view. You can change certain panes in pages of the Project section to show this view. Figure 18 shows an example:

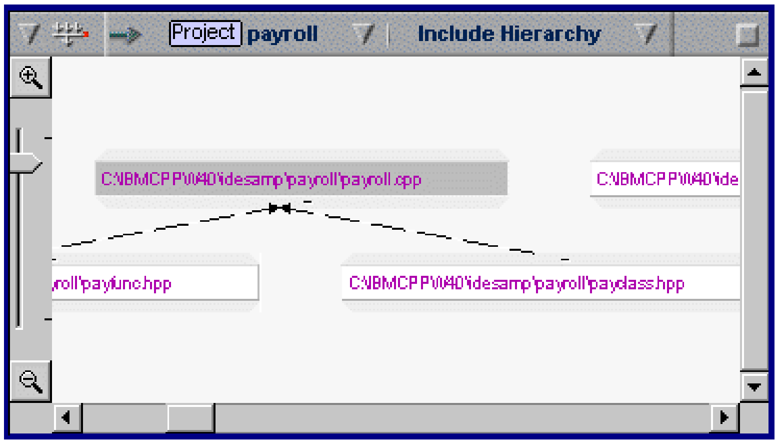

Project payroll
Include Hierarchy
C:\IBMCPPW40\idesamp\payroll\payroll.cpp
C:\IBMCPPW40\ide
roll\payfunc.hpp
C:\IBMCPPW40\idesamp\payroll\payclass.hpp

Figure 18. An Include Hierarchy view of the payroll sample

In this view, the slider and magnifying glass symbols on the left allow you to zoom in or out. This example is set to a "vertical" arrangement in the View Settings (found in the Workbook section); this means that the expandable nodes (shown as + or - symbols) expand in a vertical fashion. The arrows point to the including file.

To change the view in a pane, pull down the view type menu (see Chapter , “The view type menu” on page 96 for an illustration) and select the view you want.

15.5 Customize pages

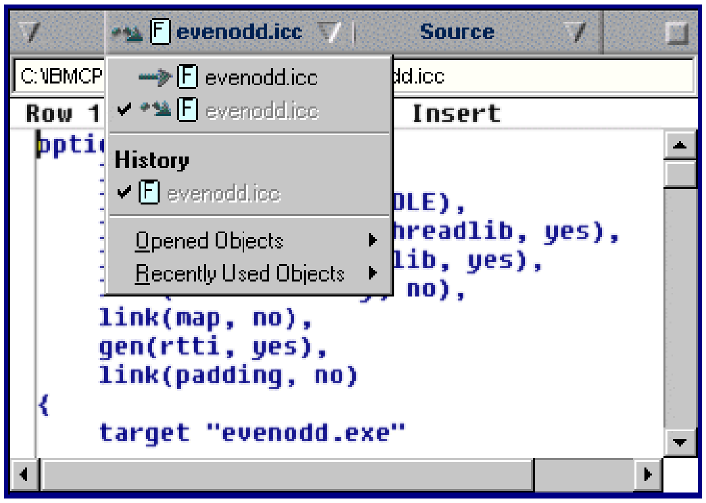
You may find there are times when the default arrangement of views in the various predesigned pages does not suit your needs. You can change the size and shape of existing panes, add or remove a pane altogether, and save all your settings.

Here is an example exercise that demonstrates the following:

- Adding a new page to a section
- Changing the linking state of a pane
- Creating a new page description

Follow these steps to add a new Source view to the File System page, so that you can compare two files from different projects, and then save the settings and add a page button for the new page layout:

1. Go to the **File System** page in the Host section, and browse to the **/idesamp/evenodd** subdirectory from the VisualAge installation directory.
2. Select any file in the **Files** view to make sure it has focus, and so that the **Source** view below will have some content. When you add a new pane, it is automatically linked to the pane that currently has focus.
3. Hold your mouse pointer over the right or left edge of the **Source** view (the **Files** view should still have focus), and hold down the **Ctrl** key.
4. When the pointer turns to a large arrow with a plus (+) sign, drag it toward the center of the window. Release the mouse when the new pane is approximately half the width of the window.
5. If the view in the window is not a Source view, pull down the view menu and select **Source**. Both Source views should be the same now.
6. In the Files view, select the evenodd configuration file (**evenodd.icc**). Both Source views should show the contents of this file.
7. Switch focus to one of the Source views. Pull down the object menu, and select the broken arrow, as shown by the first checkmark in Figure 19.



```
opti
                   OLE),
                   hreadlib, yes),
                   lib, yes),
                       no),
    link(map, no),
    gen(rtti, yes),
    link(padding, no)
{
    target "evenodd.exe"
```

Figure 19. Temporarily disconnecting the link into a pane

8. This will disconnect the link that causes the pane to update each time a new file is selected. In effect, it will "freeze" the view with evenodd.icc.
9. Now, browse through the File Tree view to a different subdirectory, for example, **/mle**. Select **mle.icc** in the Files view. You now have two Source views showing two different files. You can copy and paste between them easily. If you want to increase the viewable area in the Source views, resize them to make them deeper.
10. Now, save the new page setup. Pull down the **Page** menu, and select **Save Page Description As...**.
11. In the Save Page Description As dialog, type a name for the new page setup, such as Comparison. Make sure the radio button to **Add as New Page Description** is selected. After you click **OK**, a page button with the new page name is added to the workbook section. You can view different files in the "frozen" pane by reattaching the link, browsing to a new file, and disconnecting the link again. If you decide to remove the page later, the *page description* will remain available, and you can add it to the workbook again without repeating these steps. For more details, search for "Add a page" in the online help.

15.6 Search for objects, not for strings

As your project grows, finding specific lines of code becomes a more daunting task. Sometimes, a simple search for a function name (a string search) will yield a long list of possible matches: the function name, other names that may include the function name, and comments that mention the function. In other cases, the search results may be too narrow to locate all the uses of your object. For example, if you are looking for all the places where a class is used, you may not only be interested in finding occurrences of a specific class name.

The Find Uses search is a quick and easy way to filter your search by the type of object you are looking for, and eliminate hits on lines that do not concern you. Try the following, to see how the Find Uses function differs from the regular search function:

1. Open the payroll sample in the idesamp directory.
2. On the Search page, type `class employee` in the Search field of the Search view, and press Enter, or click the flashlight button. This search yields only one match.

   This is enough, if you only want to find the definition of the class. But what if you wanted to find classes that inherit from this class as well?
3. Switch to the Find Uses page. From the Declarations view, select the first declaration (the `employee` class declaration). The Find Uses view should show four matches: the definition of this class and the definitions of the other classes which inherit from it.
4. Now return to the Search page and search only for the string `employee`. This would be one way to use a string search to include definitions of classes that inherit from the employee class. However, this search yields many more matches: 32 in total, including every comment that mentions the class, and text strings that include the word employee.

Note: When using the Find Uses view or any Search view, including Live Find within a view, be aware that the search you perform may be restricted by two factors:

1. A filter on the view. For more detailed information on filtering, please see Appendix , “The filter menu” on page 95.
2. The scope of the object you are searching. A search in any view will only look as far as the object the view is linked to. See Appendix , “The object menu” on page 95 for instructions on changing the object viewed.

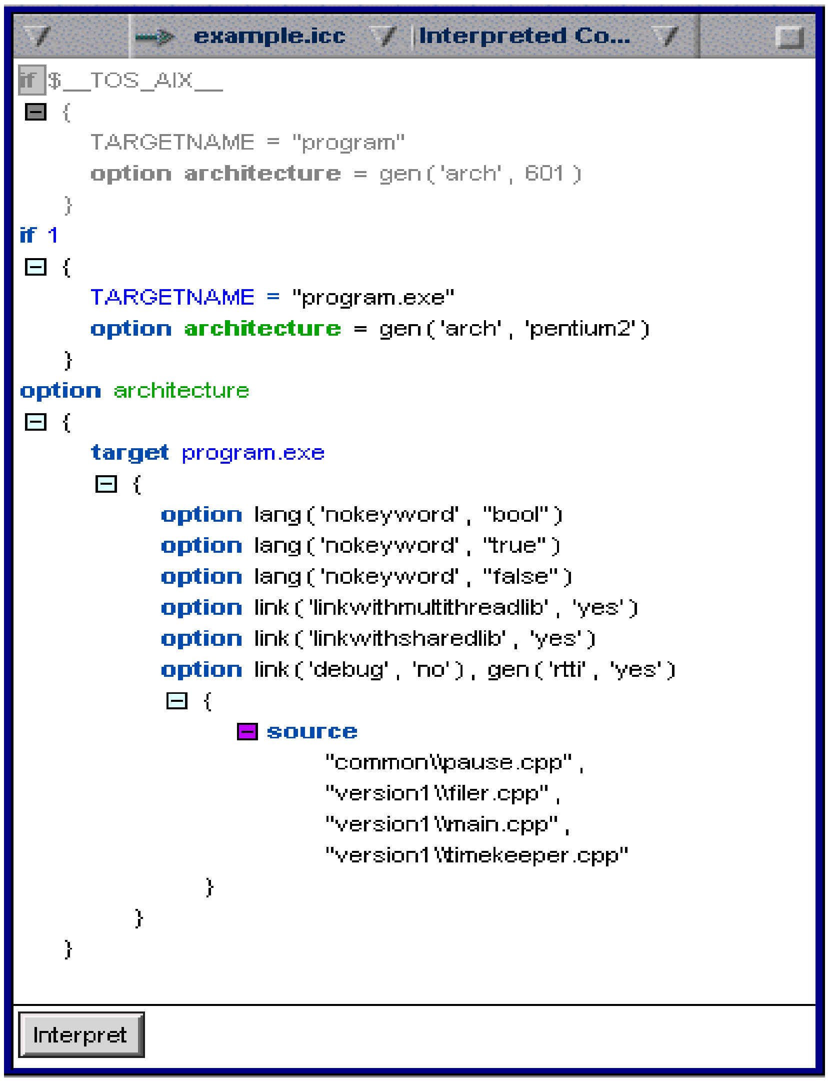
15.7 See how your configuration file was interpreted

How do you tell when you have made a mistake in your makefile? You wait for the build to fail, right? Not so with configuration files. If you have conditional processing, calls to environment variables, or variables of your own set up in your configuration file, you can see exactly how they will be evaluated without launching a build, by using the Interpreted view of the file in the Advanced page of the Configuration section.

Here is an example of an unprocessed configuration file:

```
if $__TOS_AIX__
{
       TARGETNAME="program"
       option architecture=gen(arch,601)
}

if $__TOS_WIN__
{
       TARGETNAME="program.exe"
       option architecture=gen(arch, pentium2)
}

option link_options=link(linkwithmultithreadlib, yes),
link(linkwithsharedlib, yes)

option lang_options=lang(nokeyword, "bool"),
lang(nokeyword, "true"),lang(nokeyword, "false")

option architecture, link_options, gen(rtti, yes)
   {
   target TARGETNAME
       {
       option lang_options
           {
           source "common\\pause.cpp",
           "version1\\filer.cpp",
           "version1\\main.cpp",
           "version1\\timekeeper.cpp"
           }
       }
   }
```

Figure 20 on page 89 shows how this file looks after it has been parsed. The Interpreted view resolves the `if` statements, and replaces variables with the calculated values. For example, if you are working on a Windows system, the directive block beginning with `if $__TOS_AIX__` would be ignored; this is shown in the Interpreted view as greyed-out text. The directive block beginning `if $__TOS_WIN__` evaluates to 1, or true. The options that were applied are shown in darker text, and those that were ignored are greyed out.



```
if $__TOS_AIX__
{
    TARGETNAME = "program"
    option architecture = gen ( 'arch' , 601 )
}
if 1
{
    TARGETNAME = "program.exe"
    option architecture = gen ( 'arch' , 'pentium2' )
}
option architecture
{
    target program.exe
    {
        option lang ( 'nokeyword' , "bool" )
        option lang ( 'nokeyword' , "true" )
        option lang ( 'nokeyword' , "false" )
        option link ( 'linkwithmultithreadlib' , 'yes' )
        option link ( 'linkwithsharedlib' , 'yes' )
        option link ( 'debug' , 'no' ) , gen ( 'rtti' , 'yes' )
        {
            source
                "common\\pause.cpp" ,
                "version1\\filer.cpp" ,
                "version1\\main.cpp" ,
                "version1\\timekeeper.cpp"
        }
    }
}
```

Figure 20. Interpreted ("preprocessed") configuration file

15.8 IDE Shortcuts

If you do not enjoy typing or browsing through menus to perform tasks, some of the tips in this section may be useful to you.

15.8.1 Create new configuration files

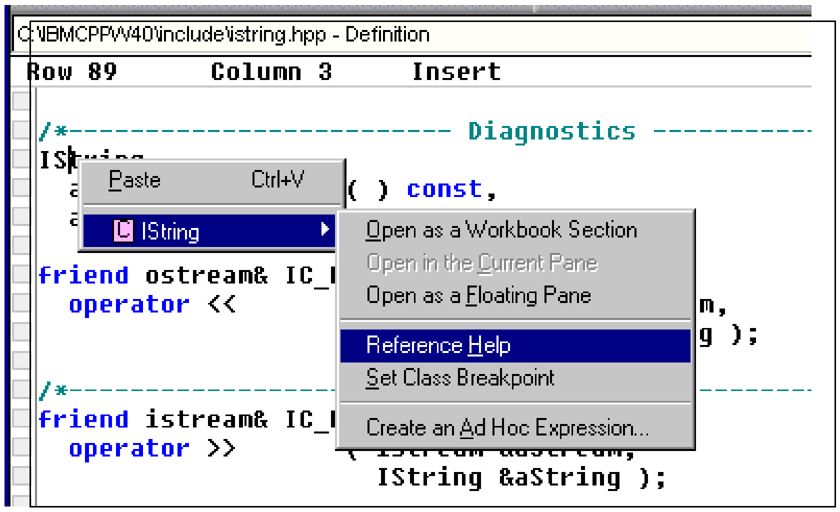
If you create configuration files by typing the directives into a text file, you must first save the file with an .icc extension, then open a project by selecting this file. However, if you use the Project SmartGuide, you will not have to type the contents, and you will have the option of having the project immediately opened for you.

To start the SmartGuide, select **Create Project...** from the **Project Workbook** menu. Based on the selections you make for your project in this wizard, options, source directives, and target directives will all be automatically added to the configuration for you. You can always tune or edit the configuration later; this is a fast way to get started.

15.8.2 Help on OpenClass classes

To access reference help on IBM Open Class classes, you do not always have to navigate from the Help Home Page. Instead, try this:

1. Select a class name or member function name in a Source view.
2. Right-click on the highlighted word to open the pop-up menu.
3. From the pop-up menu, select the class object (C) or member function (F). A second pop-up menu appears.
4. From the second menu in Figure 21, select **Reference help**. This launches the online help, and opens it to the information on this class or member function.



```
C:\IBMCPPW40\include\istring.hpp - Definition
Row 89        Column 3        Insert

/*------------------------- Diagnostics -----------
IString
                                   ( ) const,


friend ostream& IC_
  operator <<                                      m,
                                                   g );

/*---------------------------------------------------------
friend istream& IC_
  operator >>
                               IString &aString );
```

Figure 21. Finding reference help for IBM OpenClass classes

15.8.3 Live Find

Press **Ctrl+F** to open the Live Find search bar. Press **Esc** to remove it.

15.8.4 Switch pane focus

You do not need to use the mouse: F6 will move the focus to the next pane in a page.

15.8.5 Start a build

Press **Ctrl+Shift+B**.

15.8.6 Show the link diagram

If you frequently make use of the link diagram, you may find it faster to add the Show/Hide links icon to your toolbar.

Click the **Toolbar configuration** in the **Settings** page, in the Workbook section. Select the **Workbook** toolbar. In the list of available icons, highlight the **Show Links** icon, and click **Add>>**.

15.9 Make more real estate available

Getting crowded on your screen? Table 7 shows a few ways to can simplify the workbook:

Table 7. Clear up screen space in the IDE

What you can do	How to do it	How to undo it
Hide toolbar icons	Workbook section-> Settings page-> Toolbar Configuration button. Deselect the Show Toolbar checkbox for *each of the four* Toolbars.	Reset the checkbox (do not forget to pull down the list of Toolbars and reset the checkbox for each one).
Hide section tabs (note: this will also hide the page buttons)	Click on the maximize button at the far right end of the menu bar (directly below window-maximize/minimize buttons).	Click again.
Set screen fonts smaller	Workbook section-> Settings page->Default font size (select Small)	Same steps (select another size)
Hide pane title bars	Workbook section-> Settings page-> Select the **Auto hide pane title bar** checkbox	To show title bars temporarily, hold mouse pointer over top edge of the pane. To restore title bars permanently, deselect the **Auto hide pane title bar** checkbox
Fit more tabs in the workbook	Workbook section-> Settings page->Section Tab and Page Button Style->Select **Proportional**	Follow same steps; select **Fixed**.

Appendix A. Guide to interface elements

Here we supply a brief list of the elements of the Integrated Development Environment (IDE), to help you understand terms used elsewhere in this book.

A.1 The workbook

The IDE is designed to resemble a workbook, divided into *sections* by *tabs,* shown in Figure 22:

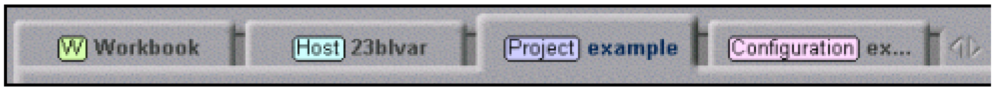


Figure 22. Tabs divide the major workbook sections

Within each section, a selection of page buttons will take you to different *pages*. While all the pages in a section show a different combination of views, they will all be *linked* to the same object. This means that all the views show a perspective on the same object: in Figure 23, for example, the object on the section tab is the host (the machine). So, pages in this section will not contain views of your project setup, or of objects contained in your code. Instead, they will contain views in which you can browse objects related to the host machine, such as directory structures and files.

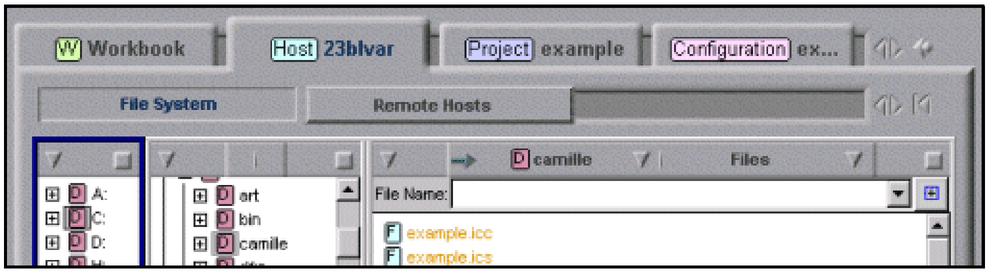


Figure 23. Pages in the Host section

Similarly, pages in the other default sections provide views related to the section:

- Pages in the Workbook section provide views for controlling the workbook itself (settings for views, font size, toolbar arrangement, and so on.)
- Pages in the Project section provide views for performing tasks related to the contents of your project (such as viewing source files or classes, and debugging programs.)
- Pages in the Configuration

New sections can be opened to show objects other than the default set, and sections you are not using can be removed and replaced easily. If you have many sections open at once, and cannot see all the tabs, you can scroll through the selection with the buttons to the right of the tabs.

You can open a new section for virtually any object in the IDE: click on the object, click mouse button 2, and select **Open as a Workbook section** from the pop-up menus. When the section is opened, one or more pages may automatically be added, but only those views relevant to the object shown on the section tab will be available.

A.2 Pages

Each workbook section offers a default selection of *page buttons* like the set shown in Figure 24:

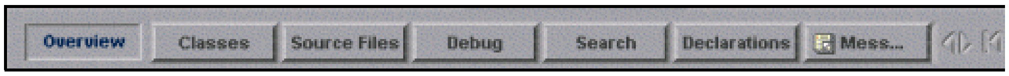


Figure 24. Page buttons available in the Project workbook section

Each page is a collection of *panes*, which contain views. The views available in each pane depend on the object the pane is linked to. In the default arrangement of views on these pages, the object shown on the workbook section tab represents the widest scope, and each of the views represents a particular focus, or narrowed scope, on that object. However, you can adjust the scope of any of the views to suit your needs.

The selection of page buttons available in each section of the workbook can be changed easily. You can design pages of your own and remove pages you do not use often. Replacing them later is easy.

A.3 Panes

There are four menus that control what appears in each pane. From left to right, they are:

The pane menu

This controls the pane itself; you can then remove the pane, rearrange the page (that is, move the panes) or to enlarge the view to fill the page (maximize the pane).See Figure 25.

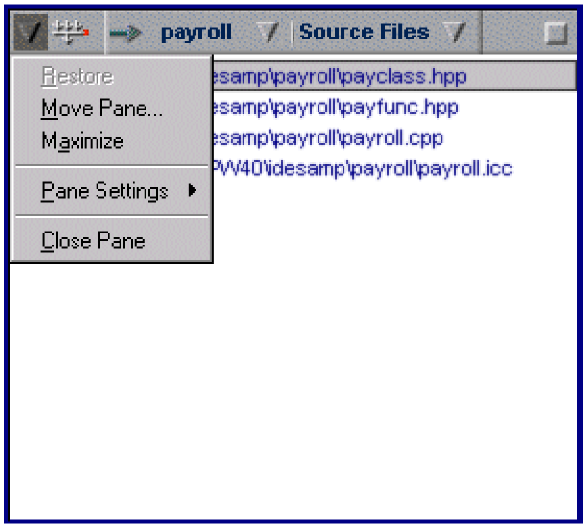

payroll
Source Files
Restore
Move Pane...
Maximize
Pane Settings
Close Pane
esamp\payroll\payclass.hpp
esamp\payroll\payfunc.hpp
esamp\payroll\payroll.cpp
PW40\idesamp\payroll\payroll.icc

Figure 25. The pane menu

The filter menu

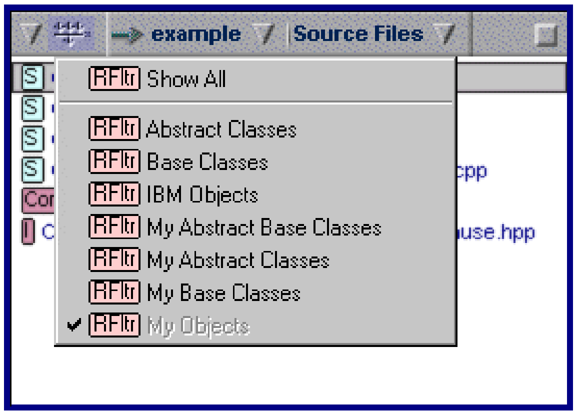


Figure 26. The filter menu

The menu shown in Figure 26 lists all the possible ways you can focus the current view. By default, all views with filters have the My Objects filter selected. This means that they will only display views of objects (source code) that you have provided; system header files are not displayed.

Note: If you perform a search in a view with any filter other than Show All applied, the scope of the search is similarly restricted. (When a red dot appears next to the filter symbol, a filter is being applied; the dot disappears when Show All is selected).

Here is an example of how filtering affects searching:

You have included the header file iostream.h in your project by issuing a #include directive in a source file, but you have not listed iostream.h directly in your configuration file. You want to search for a string that appears only in iostream.h, so you go to the Search page and are using the Search view, with the default filter (My Objects) appplied. In this case, the search will fail to find the string. "My Objects" refers only to those sources you have created. It excludes any files from the VisualAge C++installation directories.

To ensure system header files are included, you must select the Show All filter.

To see or change the criteria that define each filter, use the Filters page in the Workbook section.

The object menu

Figure 27 shows the menu that determines which object is being displayed in the view.

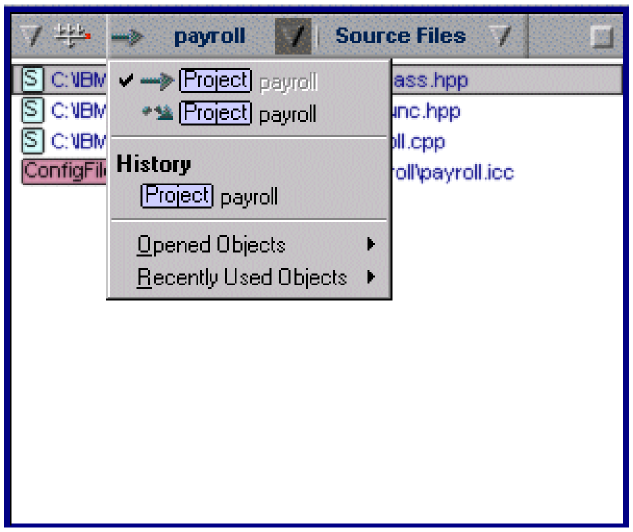


Figure 27. The object menu

Depending on which other panes the pane is linked to, the object can be as broad in scope as the project, or as granular as a function. For different types of objects, different types of views are available.

To "freeze" a pane with the view it is currently showing (for example, if you want to select a different object in another, but you don not want the contents of this pane to change), select the broken arrow from this menu (second item). This disconnects the link from other panes to this pane, until you reconnect it by selecting the first arrow.

To view a different object, you can select objects from the Opened Objects or the Recently Used Objects lists. When you do this, the link going into this pane will automatically be broken (as though you had selected the broken arrow in addition to selecting a new object). To restore the view, simply reselect the first arrow.

The view type menu

Figure 28 shows an example of the different types of views available in a pane.

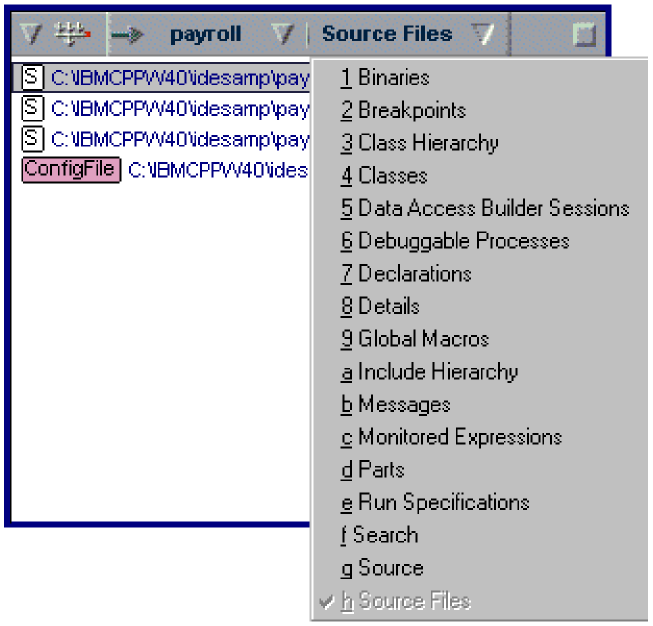


Figure 28. The view type menu

In Figure 28, list of view types is also called the view list. It is the list of views that can be applied to the object currently being viewed by this pane. Not all view types apply to all objects. Change the view by selecting the new view type from this list.

A.4 Toolbars

The row of icons at the top of the IDE window is actually a collection of several *toolbars*. A toolbar is a set of buttons for performing actions related to a specific part of the workbook.

There are four different toolbars, not all of which may appear on your screen:

- The **Workbook** toolbar, shown in Figure 29, provides buttons for project-wide actions, such as beginning or stopping a build, or opening a new project.

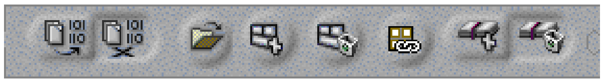


Figure 29. The workbook toolbar

- The **Pane** toolbar, shown in Figure 30, has buttons for actions related to a single pane, such as connecting links and scrolling through the history of objects that have been viewed in the pane.

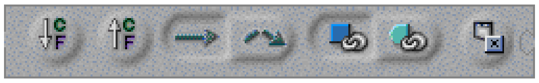


Figure 30. The pane toolbar

- The **View** toolbar, shown in Figure 31, offers a different selection of buttons, depending on the type of view in the pane that currently has focus. The buttons shown here, for editing tasks, appear when a Source view has focus. At other times, you may only see the Live Find button (the flashlight), or another combination of buttons.

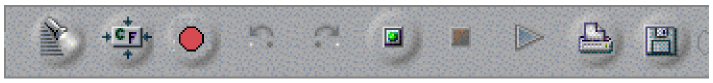


Figure 31. The view toolbar

- Finally, the **Process** toolbar, shown in Figure 32, offers debugging actions.

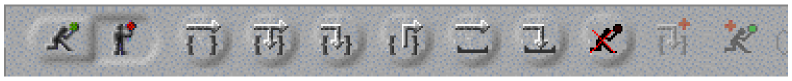


Figure 32. The process toolbar

At most times, you will only see one or two of these toolbars on your screen.

To make sure all the toolbars you want to see will display at the appropriate times, check the toolbar settings in the Workbook section. Select the Settings page, and click on the Toolbar Configuration button. In order for any toolbar to display, the Show toolbar checkbox must be checked for that toolbar. Each toolbar must be selected individually from the pull-down list at the top of the toolbar configuration window. In the current release, there is no way to select or deselect all the toolbars at once.

Appendix B. Special notices

This publication is intended to help software developers who use C++ to develop applications using IBM VisualAge C++ Professional, Version 4.0. The information in this publication is not intended as the specification of any programming interfaces that are provided by IBM VisualAge C++ Professional, Version 4.0. See the PUBLICATIONS section of the IBM Programming Announcement for IBM VisualAge C++ Professional, Version 4.0, for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

Appendix C. Related publications

See the CD-ROM references below for a more detailed discussion of the topics covered in this redbook.

C.1 Redbooks on CD-ROMs

Redbooks are also available on the following CD-ROMs. Click the CD-ROMs button at `http://www.redbooks.ibm.com/` for information about all the CD-ROMs offered, updates and formats.

CD-ROM Title	**Collection Kit Number**
System/390 Redbooks Collection	SK2T-2177
Networking and Systems Management Redbooks Collection	SK2T-6022
Transaction Processing and Data Management Redbooks Collection	SK2T-8038
Lotus Redbooks Collection	SK2T-8039
Tivoli Redbooks Collection	SK2T-8044
AS/400 Redbooks Collection	SK2T-2849
Netfinity Hardware and Software Redbooks Collection	SK2T-8046
RS/6000 Redbooks Collection (BkMgr Format)	SK2T-8040
RS/6000 Redbooks Collection (PDF Format)	SK2T-8043
Application Development Redbooks Collection	SK2T-8037
IBM Enterprise Storage and Systems Management Solutions	SK3T-3694

How to get ITSO redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.

- **Redbooks Web Site** `http://www.redbooks.ibm.com/`

  Search for, view, download, or order hardcopy/CD-ROM redbooks from the redbooks Web site. Also read redpieces and download additional materials (code samples or diskette/CD-ROM images) from this redbooks site.

  Redpieces are redbooks in progress; not all redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

- **E-mail Orders**

  Send orders by e-mail including information from the redbooks fax order form to:

	e-mail address
In United States	usib6fpl@ibmmail.com
Outside North America	Contact information is in the “How to Order” section at this site: `http://www.elink.ibmlink.ibm.com/pbl/pbl/`

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	Country coordinator phone number is in the “How to Order” section at this site: `http://www.elink.ibmlink.ibm.com/pbl/pbl/`

- **Fax Orders**

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	Fax phone number is in the “How to Order” section at this site: `http://www.elink.ibmlink.ibm.com/pbl/pbl/`

This information was current at the time of publication, but is continually subject to change. The latest information may be found at the redbooks Web site.

IBM Intranet for Employees

IBM employees may register for information on workshops, residencies, and redbooks by accessing the IBM Intranet Web site at `http://w3.itso.ibm.com/` and clicking the ITSO Mailing List button. Look in the Materials repository for workshops, presentations, papers, and Web pages developed and written by the ITSO technical professionals; click the Additional Materials button. Employees may access `MyNews` at `http://w3.ibm.com/` for redbook, residency, and workshop announcements.

IBM Redbook Fax Order Form

Please send me the following:

Title	**Order Number**	**Quantity**

First name	Last name	
Company		
Address		
City	Postal code	Country
Telephone number	Telefax number	VAT number

☐ Invoice to customer number

☐ Credit card number

Credit card expiration date	Card issued to	Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

Glossary

Assignment directive

The '=' in a configuration file. In other words, this is the directive that assigns a value to a variable.

Codestore

The database of information generated when VACPP builds a project. This includes information about the sources used to build the project, the relationships between those sources, relationships between functions, options, and debug information. The codestore is stored in an .ics file.

Configuration

The set of source files, options, variables and targets that make up a single project.

Configuration file

A text file composed of directives that control what is compiled, which sources are used, and which options and variables are applied when VACPP builds a project.

Incremental compilation

Compiling only those granular parts, such as C++ methods, that have changed or been affected by change since the previous build. This is in contrast to traditional batch compilers that compile larger parts, such as entire files.

Filter

A set of criteria applied to a view to exclude selected types of information from the view. If a filter is currently being applied to a view, a red dot will appear next to the filter indicator on the pane title bar.

See Appendix , "The filter menu" on page 95 for an illustration of the filter indicator and menu.

Macro source

A source file containing macros, and specified as such by the option macros(global,yes). (That is, the source file is listed within the { and } of the option directive.)

Macros in these sources are visible to all non-macro source files, as well as to any other primary macro sources in the same configuration file.

One-definition rule

The rule specified in the C++ standard that states:

1. A translation unit must not contain more than one definition of any variable, function, class type, enumeration type, or template.

2. A program must contain exactly one definition of every non-inline function or object that is used in that program. The definition may appear explicitly in the program or in a library, or it may be implicitly defined by the compiler (as in the case of an implicit constructor). An inline function must be defined in every translation unit in which it is used.

Option directive

The directive (the word *option*) that identifies the keyword or list of keywords that follow it as options, or as the name of a group of options.

Page description

The set of criteria that form a "template" for a page in the IDE workbook. It describes how many panes are in the page, which views are displayed in the panes, and which object the page is showing. You cannot directly edit a page description but you can modify it by making changes to the page and selecting **Save Page Description** or **Save Page Description as...** from the Page menu.

Primary source

A source file that is listed in the configuration file.

Promote (a source file)

To make a secondary source file into a primary source file by adding it to the configuration file.

Secondary source

A source file that is not listed in the configuration file, but is built as part of the project because it has been included in another source file.

Source

A source is any file used as input to a build. A source can be a C++ file, a header file, resource file, object file, and so on.

Source directive

The directive (the word source) that identifies the filename or list of filenames that follow it as input to a build.

Target

The output of a build. This can be an executable file, library, or object file.

Target directive

The directive (the word *target*) that identifies the filename that follows it as the output of a compilation.

Type clause

The specifier, of the form `type(`*`string`*`)` that can be included in a source directive in order to identify a source file as a type recognized by the compiler, for example, type(cpp) in the following source directive is the type clause:

```
source type(cpp) testfile.i
```

Target directive

The directive (the word *target*) that identifies the filename that follows it as the output of a compilation.

List of abbreviations

DLL	Dynamic Link Library
IBM	International Business Machines Corporation
ITSO	International Technical Support Organization
OS/2	Operating System/2
NT	New Technology

Index

IBM Redbooks evaluation

Getting to Know VisualAge C++ Version 4.0
SG24-5489-00

Your feedback is very important to help us maintain the quality of IBM Redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at http://www.redbooks.ibm.com
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Which of the following best describes you?
_ Customer _ Business Partner _ Solution Developer _ IBM employee
_ None of the above

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction __________

Please answer the following questions:

Was this redbook published in time for your needs? Yes___ No___

If no, please explain:

What other redbooks would you like to see published?

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

SG24-5489-00

Printed in the U.S.A.

Getting to Know VisualAge C++ Version 4.0

SG24-5489-00